2. Product Codes

Gadiel Seroussi

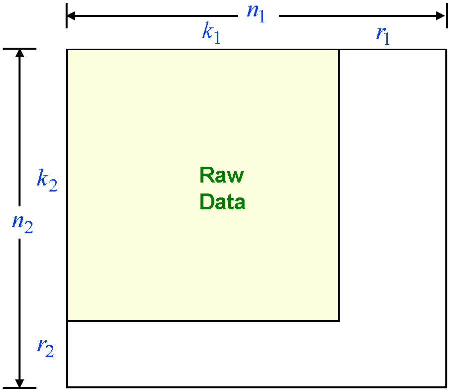
October 21, 2022

2 Product Codes

- Encoding
- Progressive redundancy
- Progressive redundancy
- Redundancy summary
- Handling random errors
- Handling random errors

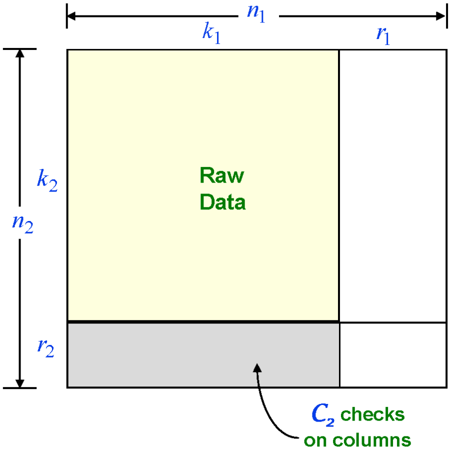
Product codes

Given $\mathcal{C}_1 : [n_1, k_1]$ and $\mathcal{C}_2 : [n_2, k_2]$, a codeword in the *product code* $\mathcal{C}_1 \times \mathcal{C}_2$ is shown in the figure (with $r_1 = n_1 - k_1, \ r_2 = n_2 - k_2$).



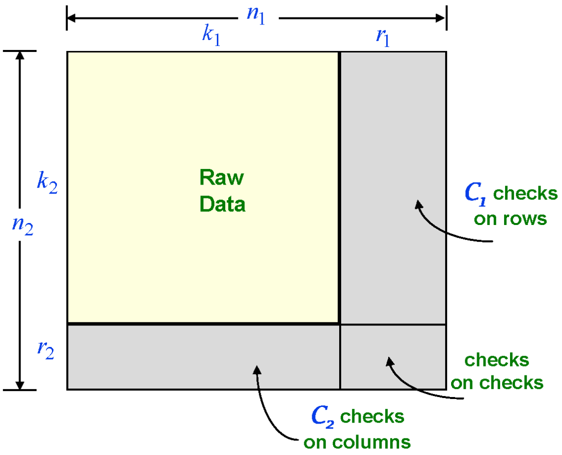
Product codes

Given $\mathcal{C}_1 : [n_1, k_1]$ and $\mathcal{C}_2 : [n_2, k_2]$, a codeword in the *product code* $\mathcal{C}_1 \times \mathcal{C}_2$ is shown in the figure (with $r_1 = n_1 - k_1,\ r_2 = n_2 - k_2$).



Product codes

Given $\mathcal{C}_1 : [n_1, k_1]$ and $\mathcal{C}_2 : [n_2, k_2]$, a codeword in the *product code* $\mathcal{C}_1 \times \mathcal{C}_2$ is shown in the figure (with $r_1 = n_1 - k_1,\ r_2 = n_2 - k_2$).

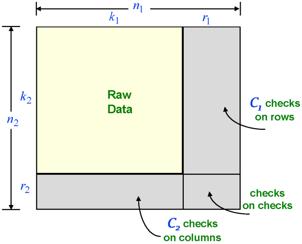


Checks on checks

- Assume product code matrix is $C \triangleq \{c_{st}\}_{s=0,\ t=0}^{n_2-1, n_1-1} = [\mathbf{c}_0, \mathbf{c}_1, \ldots, \mathbf{c}_{n_1-1}]$.

columns

- Assume we encode columns first, and let the (systematic) generator matrix of $\mathcal{C}_1$ be $[\, I_{k_1 \times k_1} | A_{k_1 \times (n_1-k_1)} \,]$. When encoding rows, a typical element in the redundancy columns is



$$c_{s,t} = \sum_{h=0}^{k_1-1} c_{s,h} A_{h,t-k_1} \,, \quad 0 \le s < n_2,\ k_1 \le t < n_1 \,,$$

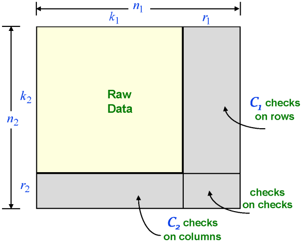
Therefore,

$$\mathbf{c}_t = \sum_{h=0}^{k_1-1} A_{h,t-k_1} \mathbf{c}_h \,.$$

Checks on checks

- Assume product code matrix is $C \triangleq \{c_{st}\}_{s=0,\ t=0}^{n_2-1,n_1-1} = [\mathbf{c}_0, \mathbf{c}_1, \ldots, \mathbf{c}_{n_1-1}]$.

columns



- Assume we encode columns first, and let the (systematic) generator matrix of $\mathcal{C}_1$ be $[\, I_{k_1 \times k_1} | A_{k_1 \times (n_1-k_1)} \,]$. When encoding rows, a typical element in the redundancy columns is

$$c_{s,t} = \sum_{h=0}^{k_1-1} c_{s,h} A_{h,t-k_1} \ , \quad 0 \le s < n_2,\ k_1 \le t < n_1 \ ,$$

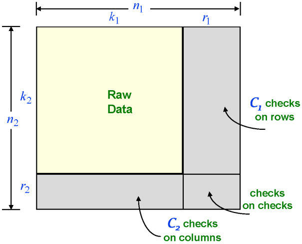
Therefore,

$$\mathbf{c}_t = \sum_{h=0}^{k_1-1} A_{h,t-k_1} \mathbf{c}_h \ .$$

- Redundancy columns are linear combinations of codewords in $\mathcal{C}_2 \Rightarrow$ they too are codewords in $\mathcal{C}_2$.

- Assume product code matrix is $C \triangleq \{c_{st}\}_{s=0,\ t=0}^{n_2-1,n_1-1} = [\mathbf{c}_0, \mathbf{c}_1, \ldots, \mathbf{c}_{n_1-1}]$.

  columns



- Assume we encode columns first, and let the (systematic) generator matrix of $\mathcal{C}_1$ be $[\, I_{k_1 \times k_1} | A_{k_1 \times (n_1-k_1)} \,]$. When encoding rows, a typical element in the redundancy columns is

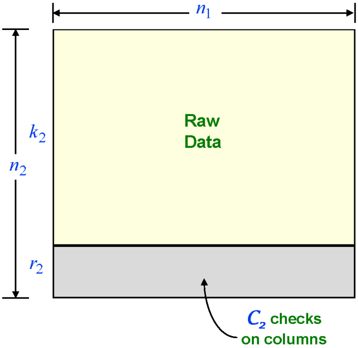
$$c_{s,t} = \sum_{h=0}^{k_1-1} c_{s,h} A_{h,t-k_1} \,, \quad 0 \le s < n_2,\ k_1 \le t < n_1 \,,$$

  Therefore,

$$\mathbf{c}_t = \sum_{h=0}^{k_1-1} A_{h,t-k_1} \mathbf{c}_h \,.$$

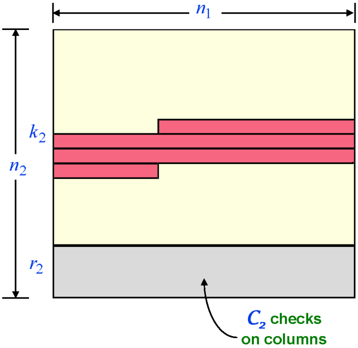
- Redundancy columns are linear combinations of codewords in $\mathcal{C}_2 \Rightarrow$ they too are codewords in $\mathcal{C}_2$.
- "Checks on checks" satisfy both the $\mathcal{C}_1$ and $\mathcal{C}_2$ constraints.
  - They are uniquely determined by the "checks on columns" region and *also* by the "checks on rows" region $\Longrightarrow$ *they are the same regardless of whether columns or rows are encoded first.*

Code interleaving



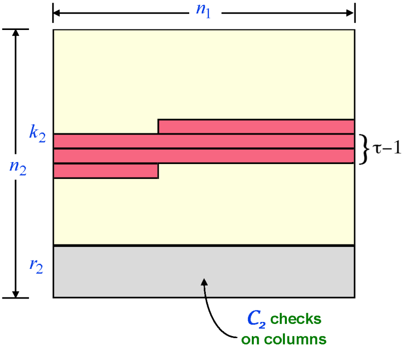
- Special case of a product code with $k_1 = n_1$ (no redundancy on rows).

Code interleaving



- Special case of a product code with $k_1 = n_1$ (no redundancy on rows).
- Useful for correcting *burst errors* (bursts run in the *row* direction).

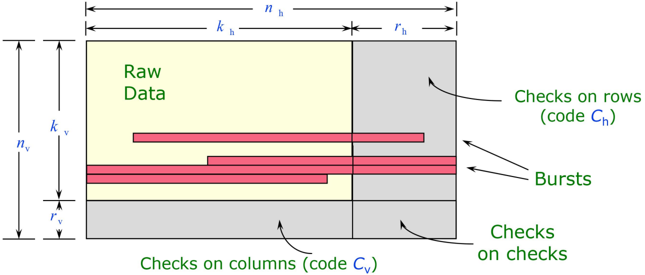
Code interleaving



- Special case of a product code with $k_1 = n_1$ (no redundancy on rows).
- Useful for correcting *burst errors* (bursts run in the *row* direction).
- Can correct any burst of length $\leq n_1\tau = n_1\lfloor(d_2-1)/2\rfloor$ using straightforward error correction of columns with $\mathcal{C}_2$

Burst correction with product codes

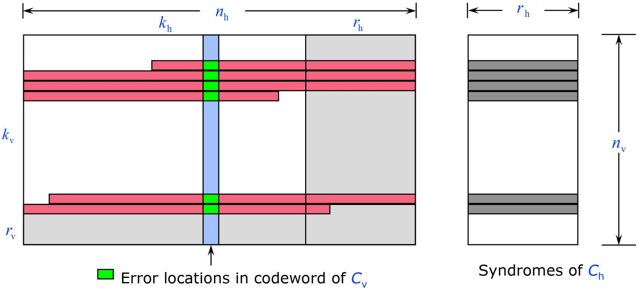
- $\mathcal{C}_h$: $[n_h, k_h = n_h - r_h]$, $\mathcal{C}_v$: $[n_v, k_v = n_v - r_v]$
- Product code $\mathcal{C}_h \times \mathcal{C}_v$:



- Overall redundancy $R = r_h n_v + r_v n_h - r_h r_v$.
- $\mathcal{C}_h$ and $\mathcal{C}_v$ assumed to be MDS codes (e.g. GRS).
- Data sent through a *bursty* channel *row by row*.

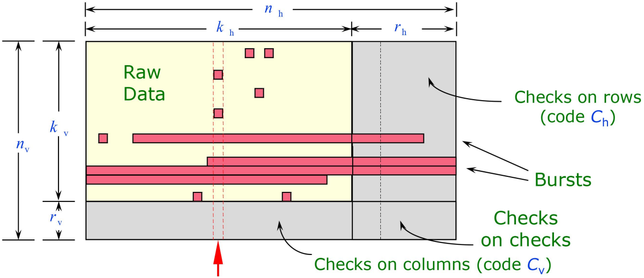
A decoding strategy for bursts

- Use $\mathcal{C}_h$ to *detect* corrupted rows, mark as *erased*.
- Use $\mathcal{C}_v$ to correct *errors and erasures*, using the location information provided by $\mathcal{C}_h$.



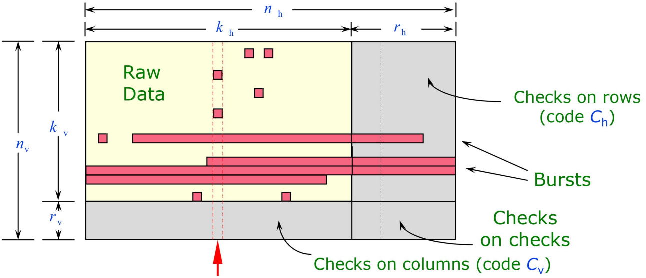
- Choose
  - r_h so that Prob($\mathcal{C}_h$ misses a corrupted row) ($\propto q^{-r_h}$) is "small enough."
  - r_v so that Prob(more than r_v corrupted rows) is "small enough."

How about random errors?



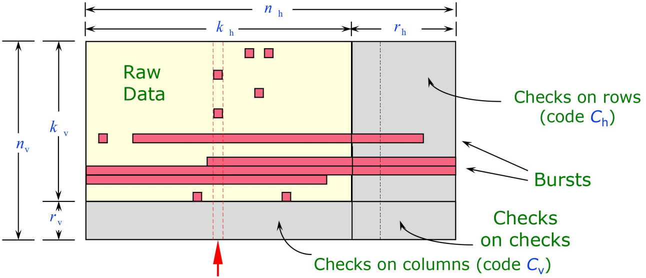
- Use part of the redundancy of $\mathcal{C}_h$ to attempt correction. In the figure, the marked column may be uncorrectble by $\mathcal{C}_v$ alone.

How about random errors?



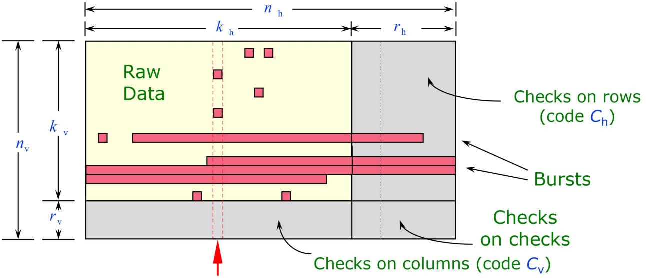
- Use part of the redundancy of $\mathcal{C}_h$ to attempt correction. In the figure, the marked column may be uncorrectble by $\mathcal{C}_v$ alone.
- Residual redundancy in $\mathcal{C}_h$ should be sufficient to correct bursts.
- An iterative, GMD-like procedure can be used.

How about random errors?



- Use part of the redundancy of $\mathcal{C}_h$ to attempt correction. In the figure, the marked column may be uncorrectble by $\mathcal{C}_v$ alone.
- Residual redundancy in $\mathcal{C}_h$ should be sufficient to correct bursts.
- An iterative, GMD-like procedure can be used.
- Can be useful in *distributed* storage where rows are *local* and columns are *global* (distributed). Random errors are handled locally.

How about random errors?

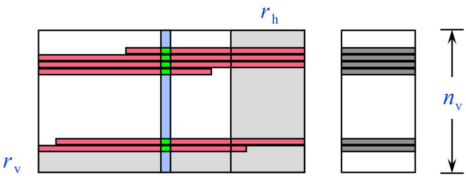


- Use part of the redundancy of $\mathcal{C}_h$ to attempt correction. In the figure, the marked column may be uncorrectble by $\mathcal{C}_v$ alone.
- Residual redundancy in $\mathcal{C}_h$ should be sufficient to correct bursts.
- An iterative, GMD-like procedure can be used.
- Can be useful in *distributed* storage where rows are *local* and columns are *global* (distributed). Random errors are handled locally.
- We will focus on burst-only correction for now, will get back to random errors at the end.

Probabilistic decoding

- *Assumption:* Received data in burst region is uniformly distributed over $GF(q)$.
- Decoding *does not guarantee* correction of all error patterns affecting r_v rows or less. For that, redundancy $\geq 2n_h r_v$ would be required.
- Instead, we allow a *small probability* ($\propto q^{-r_h}$) of missing a pattern of $\leq r_v$ rows.

Why is this scheme inefficient?



- $\mathcal{C}_h$ uses r_h check symbols for *each* row to determine whether the row is corrupted.
- That way, $\mathcal{C}_h$ can inform $\mathcal{C}_v$ about *any* combination of up to n_v corrupted rows.
- But $\mathcal{C}_v$ can correct only up to r_v erasures $\Longrightarrow$ it can only handle up to r_v corrupted rows!
- Information about combinations of $r_v + 1$ or more corrupted rows is useless for $\mathcal{C}_v$.
- *But we are paying for that information ...*

Reference

IEEE TRANSACTIONS ON INFORMATION THEORY, VOL. 44, NO. 4, JULY 1998 1395

Reduced-Redundancy Product Codes for Burst Error Correction

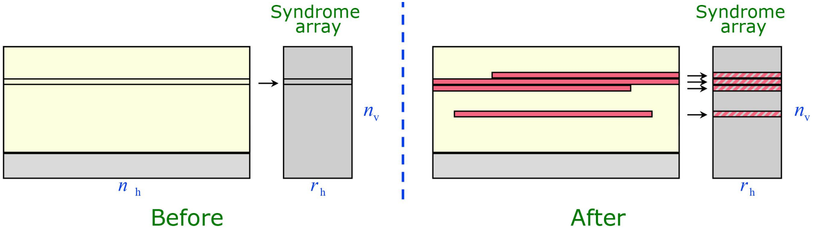
Ron M. Roth and Gadiel Seroussi

***Abstract*— In a typical burst error correction application of a product code of $n_{\mathrm{v}} \times n_{\mathrm{h}}$ arrays, one uses an $[n_{\mathrm{h}}, n_{\mathrm{h}} - r_{\mathrm{h}}]$ code $\mathcal{C}_{\mathrm{h}}$ that detects corrupted rows, and an $[n_{\mathrm{v}}, n_{\mathrm{v}} - r_{\mathrm{v}}]$ code $\mathcal{C}_{\mathrm{v}}$ that is applied to the columns while regarding the detected corrupted rows as erasures. Although this conventional product code scheme offers very good error protection, it contains excessive redundancy, due to the fact that the code $\mathcal{C}_{\mathrm{h}}$ provides the code $\mathcal{C}_{\mathrm{v}}$ with information on many error patterns that exceed the correction capability of $\mathcal{C}_{\mathrm{v}}$. In this work, a coding scheme is proposed in which this excess redundancy is eliminated, resulting in significant savings in the overall redundancy compared to the conventional case, while offering the same error protection. The redundancy of the proposed scheme is $n_{\mathrm{h}} r_{\mathrm{v}} + r_{\mathrm{h}}(\ln r_{\mathrm{v}} + O(1)) + r_{\mathrm{v}}$, where the parameters r_{h} and r_{v} are close in value to their counterparts in the conventional case, which has redundancy $n_{\mathrm{h}} r_{\mathrm{v}} + n_{\mathrm{v}} r_{\mathrm{h}} - r_{\mathrm{h}} r_{\mathrm{v}}$. In particular, when the codes $\mathcal{C}_{\mathrm{h}}$ and $\mathcal{C}_{\mathrm{v}}$ have the same rate and $r_{\mathrm{h}} \ll n_{\mathrm{h}}$, the redundancy of the proposed scheme is close to one-half of that of the conventional product code counterpart. Variants of the scheme are presented for channels that are mostly bursty, and for channels with a combination of random errors and burst errors.**

***Index Terms*— Array codes, generalized concatenated codes, product codes, superimposed codes.**

A coding scheme that eliminates "redundant" redundancy

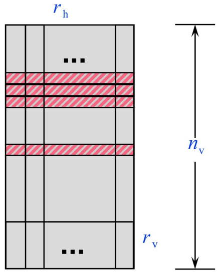
- Forget (for the time being) the check symbols of $\mathcal{C}_h$.
- For each row, compute a syndrome with respect to $\mathcal{C}_h$ (as if the row was a "received word"), forming a *syndrome array*.



- Comparing the syndromes before and after the channel, each syndrome *changed* corresponds to a corrupted row.
- In each column of the syndrome array, the number of "errors" is at most the number of corrupted rows in the main array.

Using the syndrome array to locate error patterns

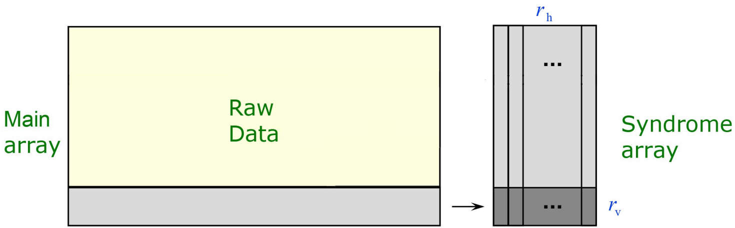
- Suppose every column in the syndrome array is a code block in an ECC ($\mathcal{C}_0$) capable of correcting r_v errors.
- Then, we can locate up to r_v corrupted rows.
- As before, r_h is chosen so that the misdetection probability of a row ($\propto q^{-r_h}$) is small enough.



Syndrome array

- How do we make the columns of the syndrome array codewords in the required ECC?
  - We need a redundancy of $2r_v$ to correct r_v errors in each column $\Longrightarrow$ we need a total of $2r_hr_v$ check symbols in the syndrome array.
  - But, with our current assumptions, we have no freedom: the syndrome array is completely determined by the main array ...

How to get the needed redundancy in the syndrome array



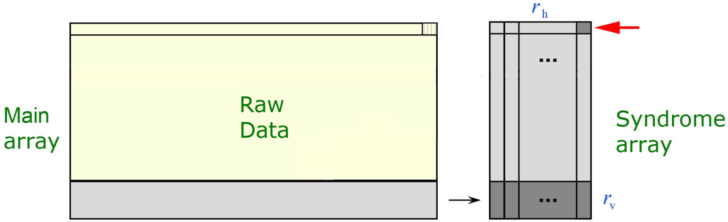
- The columns of the syndrome array are linear combinations of codewords in $\mathcal{C}_v$ $\Rightarrow$ they are codewords of $\mathcal{C}_v$ $\Rightarrow$ each contains redundancy r_v, for a total of $r_h r_v$ in the array.
- We need $r_h r_v$ more.

Review: Useful properties of GRS codes

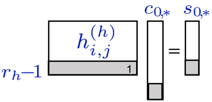
$$H = \begin{pmatrix} 1 & 1 & \dots & 1 \\ \alpha_1 & \alpha_2 & \dots & \alpha_n \\ \alpha_1^2 & \alpha_2^2 & \dots & \alpha_n^2 \\ \vdots & \vdots & \vdots & \vdots \\ \alpha_1^{r'-1} & \alpha_2^{r'-1} & \dots & \alpha_n^{r'-1} \\ \vdots & \vdots & \vdots & \vdots \\ \alpha_1^{r-1} & \alpha_2^{r-1} & \dots & \alpha_n^{r-1} \end{pmatrix}$$

- GRS codes are *nested*. The code with redundancy r' contains the code with redundancy $r > r'$.
- $\mathcal{C}_0$ will be a *subcode* of $\mathcal{C}_v$, obtained by adding r_v parity checks to the r_v already in $\mathcal{C}_v$.
- Systematic parity-check matrix: $H_{\text{sys}} = [\,A \,|\, I_{r\times r}\,]$. Here, the last r coordinates of the code are parity checks. However, *any* subset of $r = n - k$ coordinates can be taken as parity check symbols.
- If i is chosen as a parity check location then we can write $c_i + \sum_{j\neq i} h_{j,i} c_j = 0$.

Imposing a redundancy check on the syndrome array

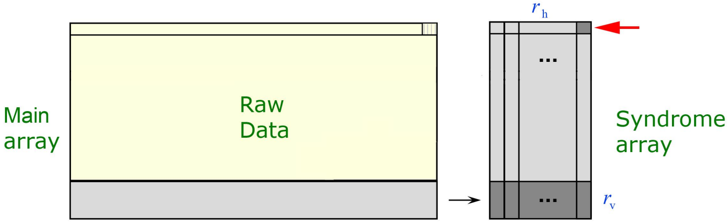


- For the marked symbol in the syndrome array, from $\mathcal{C}_h$:



$$s_{0,r_h-1} = c_{0,n_h-1} + \sum_{j=0}^{n_h-2} h^{(h)}_{r_h-1,j} c_{0,j} \,. \qquad (*)$$

Imposing a redundancy check on the syndrome array



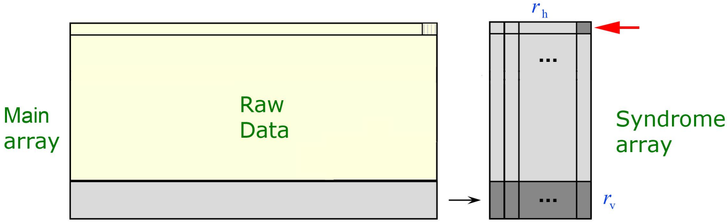
- For the marked symbol in the syndrome array, from $\mathcal{C}_h$:

$$s_{0,r_h-1} = c_{0,n_h-1} + \sum_{j=0}^{n_h-2} h^{(h)}_{r_h-1,j} c_{0,j} \,. \qquad (*)$$

- Say we want to impose an additional parity check on the syndrome array

$$s_{0,r_h-1} + \sum_{j=1}^{n_v-1} h^{(0)}_{0,j} s_{j,r_h-1} = 0 \,. \qquad (**)$$

Imposing a redundancy check on the syndrome array



- For the marked symbol in the syndrome array, from $\mathcal{C}_h$:

$$s_{0,r_h-1} = c_{0,n_h-1} + \sum_{j=0}^{n_h-2} h^{(h)}_{r_h-1,j} c_{0,j} \,. \qquad (*)$$

- Say we want to impose an additional parity check on the syndrome array

$$s_{0,r_h-1} + \sum_{j=1}^{n_v-1} h^{(0)}_{0,j} s_{j,r_h-1} = 0\,. \qquad (**)$$

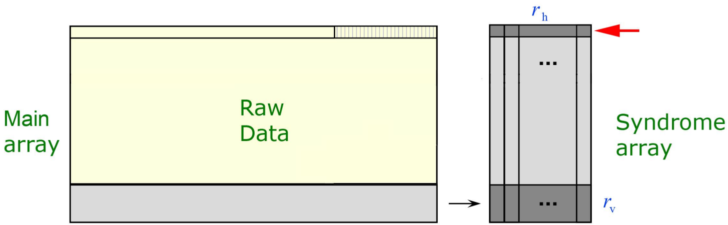
- Plugging s_{0,r_h-1} from $(*)$ in $(**)$

linear function of $c_{i,j}$, $i > 0$

$$c_{0,n_h-1} + \sum_{j=0}^{n_h-2} h^{(h)}_{r_h-1,j} c_{0,j} + \sum_{j=1}^{n_v-1} h^{(0)}_{0,j} s_{j,r_h-1} = 0\,.$$

Equivalent to imposing a parity check on c_{0,n_h-1}.

Imposing redundancy checks on the syndrome array

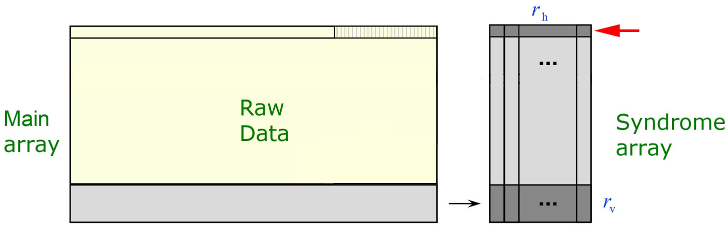


$$c_{0,n_h-1} + \sum_{j=0}^{n_h-2} h^{(h)}_{r_h-1,j} c_{0,j} + \sum_{j=1}^{n_v-1} h^{(0)}_{0,j} s_{j,r_h-1} = 0\,.$$

- Extends similarly to a full row of the syndrome array (imposing the *same* parity check constraint).

Equivalent to imposing parity checks on $c_{0,n_h-r_h} \ \cdots \ c_{0,n_h-1}$.

Imposing redundancy checks on the syndrome array



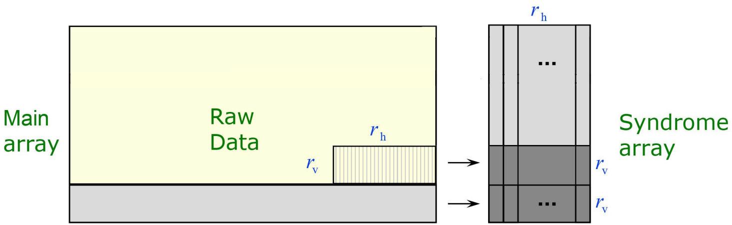
$$c_{0,n_h-1} + \sum_{j=0}^{n_h-2} h^{(h)}_{r_h-1,j} c_{0,j} + \sum_{j=1}^{n_v-1} h^{(0)}_{0,j} s_{j,r_h-1} = 0\,.$$

- Extends similarly to a full row of the syndrome array (imposing the *same* parity check constraint).

  Equivalent to imposing parity checks on $c_{0,n_h-r_h} \cdots c_{0,n_h-1}$.

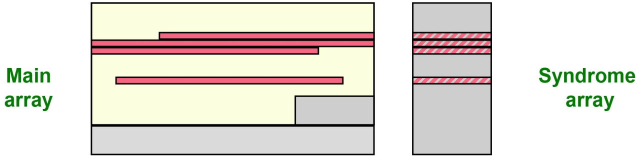
- Extends to several rows of the syndrome array (imposing a different parity check constraint for each row). The row locations are arbitrary, except for the last r_v rows, which are already taken.

How to get the needed redundancy in the syndrome array



- The additional required redundancy $r_h r_v$ can be placed in the $r_h r_v$ shaded entries.
- Total redundancy $R' = r_v n_h + r_h r_v$ ($\approx rn$), compared with $R = r_h n_v + r_v n_h - r_h r_v$ ($\approx 2rn$) in conventional product codes ($r \ll n$).

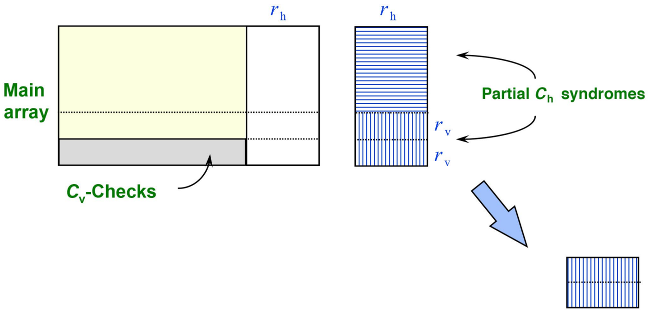
Decoding



- Upon receipt of a possibly corrupted $n_h \times n_v$ array.
  - Use $\mathcal{C}_h$ on the rows of the main array to compute the syndrome array.
  - Use $\mathcal{C}_0$ to locate up to r_v errors in each column of the syndrome array. This gives the locations of the corrupted rows of the main array. Declare those rows erased.
  - Use $\mathcal{C}_v$ to correct up to r_v erasures in the columns of the main array.

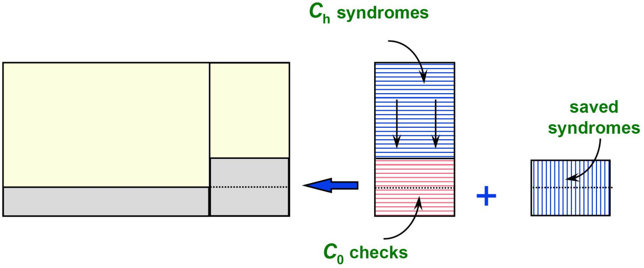
Encoding

- Stage 1:
  - Compute $\mathcal{C}_v$ checks for the first $n_h - r_h$ columns.
  - Accumulate partial $\mathcal{C}_h$ syndromes for the corresponding partial rows.
  - Save the last $2r_v$ rows of partial $\mathcal{C}_h$ syndromes.



Encoding

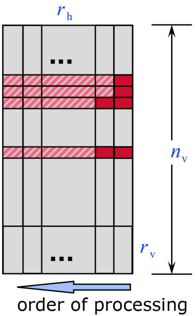
- Stage 2:
  - Complete $\mathcal{C}_h$ syndromes for the first $n_v - 2r_v$ rows.
  - Compute $\mathcal{C}_0$ checks for the above $\mathcal{C}_h$ syndromes.
  - Add the computed $\mathcal{C}_0$ checks to the saved partial syndromes from Stage 1, and store in coded array.

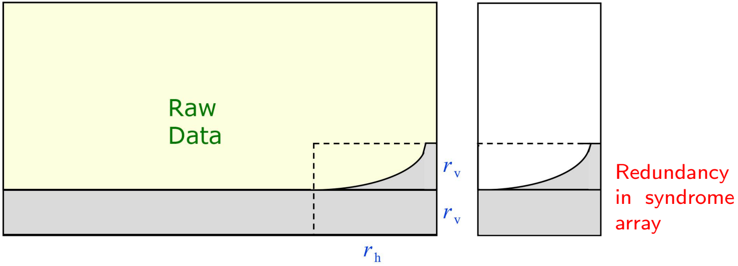


Progressive redundancy

Additional redundancy reduction:

- Decode the columns of the syndrome array one by one.
- Errors located in the first column can be marked as *erasures* in the second column $\Rightarrow$ *second column needs less redundancy*.
- Similarly for the rest of the columns.





Progressive redundancy

Components:

- $\mathcal{C}_h$ satisfying the *MDS supercode property*:

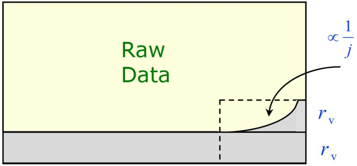
Parity check matrix of $\mathcal{C}_h$

MDS for all $1 \le j \le r_h$

- A nested family of $r_h + 1$ "vertical" codes

$$\mathcal{C}_0 \subset \mathcal{C}_1 \subset \cdots \subset \mathcal{C}_j \subset \cdots \subset \mathcal{C}_{r_h} = \mathcal{C}_v \,.$$

- Redundancies of the $\mathcal{C}_j$:
  - $r_j = r_v + \lceil r_h/j \rceil - 1$ but not greater than $2r_v$.
  - designed to *minimize redundancy* for a given *miscorrection probability*.



Total redundancy:
$R'' \le r_v n_h + r_h(\ln r_v + O(1)) + r_v \,.$

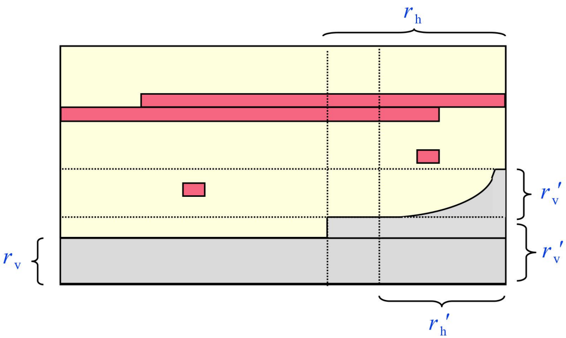
Redundancy summary

Scheme	*Redundancy*
Conventional	$r_v n_h + r_h n_v - r_h r_v$
Constant redundancy	$r_v n_h + r_h r_v$
Progressive redundancy	$r_v n_h + r_h(\ln r_v + O(1)) + r_v$

- Example: for a so-called "cut-off row-error channel," with $\text{Prob}(10 - \text{row burst}) = 10^{-3}$, targeting $\text{Prob}(\text{array error}) = 10^{-17}$. Parameters: $n_h = 96$, $n_v = 128$, $r_v = 10$.

Scheme	*Redundancy*
Conventional	1786 ($r_h = 7$)
Constant redundancy	1030 ($r_h = 7$)
Progressive redundancy	986 ($r_h = 8$)

Handling random errors



- Add *explicit* redundancy for $\mathcal{C}_h$
  - handles combined burst and random errors.

Handling random errors

- Assumption: In addition to burst errors, we handle at most s random errors in an array, with at most t in each row.
- Strategy:
  - Increase the redundancy of $\mathcal{C}_h$ by $2t$.
  - Increase the redundancy of $\mathcal{C}_0$ by $2s$.
  - Correct $r_v + s$ errors with $\mathcal{C}_0$, and t errors per corrupted row with $\mathcal{C}_h$.
- The increased redundancy of $\mathcal{C}_0$ may result in increased decoder hardware complexity. Possible trade-offs of complexity vs. redundancy are:
  - Reduce the parameters n_h and n_v, to decrease the values of s, t, and r_v.
  - Handle random errors with "explicit" redundancy in $\mathcal{C}_h$, as in conventional product codes.