# Notes for Lecture 2

## 2 Outline

Today's lecture will cover

- Another View of Reed-Solomon Codes
- Code Concatenation
- Justesen Codes

### 2.1 Another View of Reed-Solomon Codes

We will now consider a different view of Reed-Solomon codes. By selecting the appropriate "change of basis" over our finite field we will develop a simple quasi-linear time encoding scheme[1]. This presentation emphasizes the effect a choice of basis has on algorithms over finite fields, giving us a preview of the Fast Fourier Transform.

Let $\mathbb{F}_q$ be a finite field; recall that $q = p^k$ for a prime $p$ and some integer $k > 0$. We will consider Reed-Solomon codes with $n = q - 1$. In the multiplicative group $\mathbb{F}_q^*$ there exists an element $\omega \neq 0$ such that

$$\omega, \omega^2, \omega^{q-1}$$

are all distinct and $\omega^{q-1} = 1$. The elements $\omega^i$ are called *roots of unity*, and $\omega$ is the *n'th root of unity.*

The main idea of our new view of Reed-Solomon is to pick the reference points $x_1 \ldots x_n$ of the Reed-Solomon code to be roots of unity, namely $\omega, \omega^2, \ldots \omega^n$. Recall that the Reed-Solomon polynomial $p(x)$ is given by

$p(x) = c_0 + c_1 x + \ldots + c_{k-1} x^{k-1}$. The encoding of the message $c_0, \ldots c_{k-1}$ is then the values

$$p(\omega), p(\omega^2), \cdots, p(\omega^{k-1})$$

In the last lecture, we moved between the representation of codewords as coefficients and as values of a polynomial. Let's do that again for codewords with this basis. The alternate encoding of the message $c_0 \ldots c_{k-1}$ is the coefficients of the product polynomial $p(x)g(x)$, where $g(x)$ is defined as follows:

$$g(x) = \Pi_{i=1}^{n-k}(x - w^i)$$

[1] Recall that "quasi-linear" means linear up to logarithmic factors, e.g. $n \log n$ is quasi-linear.

It turns out that there exist very efficient algorithms for multiplying polynomials when $x$ is known to be a root of unity. These algorithms allow us to compute $p(x)g(x)$ in the quasi-linear time. We will revisit the topic of fast polynomial multiplication in the next lecture.

Once $p(x)g(x)$ is computed, we can simply read off the coefficients, giving us the desired quasi-linear encoding algorithm. Decoding occurs as before.

We now need to prove that this new encoding procedure produces the same Reed-Solomon codewords that we constructed in the previous lecture. More formally, let the encoding scheme from the previous lecture be denoted as $RS(c_0, \ldots, c_{k-1})$ and the one we have just described as $RS'(c'_0, \ldots, c'_{k-1})$. Then we want to show

$$\forall c_0, c_1, \ldots, c_{k-1}$$
$$\exists c'_0, c'_1, \ldots, c'_{k-1}$$

such that

$$RS(c_0, \ldots, c_{k-1}) = RS'(c'_0, \ldots, c'_{k-1})$$

We know that $RS'(c'_0, \ldots, c'_{k-1})$ is given by the coefficients of $p(x)g(x)$. Now let's write down the definition of $RS$. Define $a_i = \Sigma_{j=0}^{k-1} c_j (\omega^i)^j$. Then let $q(x) = \Sigma_{i=0}^{n-1} a_i x^i$. We then want to show that $q(x)$ is divisible by $g(x)$.

To show $g(x)$ divides $q(x)$, first note that the roots of $g(x)$ are exactly the roots of unity $\omega^i$ for $i = 1, \ldots, n-k$. Therefore it is enough to show that $q(\omega^i) = 0, i = 1, \ldots, n-k$, implying that every root of $g(x)$ is a root of $q(x)$.

Let's write down the expression for $q(\omega^i)$.

$$q(\omega^j) = \Sigma_{i=0}^{n-1} \left( \Sigma_{\ell=0}^{k-1} c_\ell (\omega^i)^\ell \right) \cdot (\omega^i)^j$$

Multiplying the $\omega$ terms gives us

$$= \Sigma_{i=0}^{n-1} \Sigma_{\ell=0}^{k-1} c_\ell \omega^{(\ell+j)^i}$$

Then we switch the order of summation and pull out the $c_\ell$ to find

$$= \Sigma_{\ell=0}^{k-1} c_\ell \Sigma_{i=0}^{n-1} (\omega^{(\ell+j)})^i$$

Notice when $j \leq n-k$, we have $\ell + j < n$. We claim that in this case the sum $\Sigma_{i=0}^{n-1} (\omega^{(\ell+j)})^i = 0$, which implies

$$q(\omega^j) = \Sigma_{\ell=0}^{k-1} c_\ell \cdot 0 = 0$$

We will now need Fermat's Little Theorem, adapted to general finite fields.

**Theorem 1** *Fermat's Little Theorem*

*Let $\mathbb{F}_q^*$ be the multiplicative group of a finite field. For any $a \in \mathbb{F}_q^*$, with $a \neq 1$, we have $a^i$ distinct for $i = 1, \ldots, q-1$ and $a^{q-1} - 1 = 0$.*

PROOF: Omitted. For details, see Lidl and Niederreiter[**?**]. □

Recall that $\ell+j < n$ and we set $n = q-1$. Therefore we know $\omega^{(\ell+j)} \neq 1$. Let's consider the polynomial $(x^n - 1)$, which by Fermat's Little Theorem has $n$ roots. Clearly $x = 1$ is a root, so we can rewrite the polynomial as $(x-1)(\Sigma_{i=0}^{n-1} x^i)$. What are the other roots? Again applying Fermat's Little Theorem, we know that all other $a \neq 1$ are also roots. Now because we know that $\omega^{(\ell+j)} \neq 1$, we obtain that $\Sigma_{i=0}^{n-1}(\omega^{(\ell+j)})^i = 0$, because substituting we find $(\omega^{(\ell+j)} - 1)(\Sigma_{i=0}^{n-1}(\omega^{(\ell+j)})^i = 0$ and $(\omega^{(\ell+j)} - 1) \neq 0$.

This shows that $q(\omega^i) = 0$ for $i = 1, \ldots, n-k$, establishing that $g(x)$ divides $q(x)$ as desired.

## 2.2 Code Concatenation

From Reed-Solomon codes, we have very nice $[n, k, d]_q$ codes for any $q = p^k$. In some applications, however, we may find it more convenient to work with a smaller alphabet; for example, we may want to set $q = 2$ or $q = 2^8$. We can construct such codes by setting $q$ to these values, but this restricts our $n$ and $k$. Alternatively, we could map elements of the finite field $\mathbb{F}_q$ to binary strings, but this loses us a factor of $\lceil \log q \rceil$ in $n$. Neither of these two ideas is satisfactory in the general case.

We now introduce our main tool for "converting" codes over one alphabet to codes over another alphabet: *code concatenation*. The key idea is that we can create a new code by *composing an existing code with another code that is over the desired alphabet.*

More formally, suppose we have two error-correcting codes $C_i$ and $C_o$ such that

$$\begin{aligned} C_o &: \Sigma_o^{k_o} \to \Sigma_o^{n_o} \\ C_i &: \Sigma_i^{k_i} \to \Sigma_i^{k_o} \end{aligned}$$

and further, there exists a polynomial time computable $1-1$ mapping $\Sigma_i^{k_2} \leftrightarrow \Sigma_o$. We call $C_i$ the *inner code* and $C_o$ the *outer code*. We then *concatenate* the inner code and outer code by composing their encoding and decoding algorithms[2]. The resulting concatenated code maps messages over the alphabet $\Sigma_o$ to codewords in the alphabet $\Sigma_i$.

For encoding, we first apply the outer encoding algorithm $C_o$ to a message in $\Sigma_o^{k_o}$. This produces a codeword in $\Sigma_o^{n_o}$. For each of the $n_o$ symbols in the codeword, we map the symbol to an element of $\Sigma_i^{k_i}$ and apply the encoding algorithm for $C_i$ to obtain an element of $\Sigma_i^{n_i}$. The result is a message in $\Sigma_i^{n_o n_i}$.

Decoding is just the opposite. First we interpret the codeword as a sequence of elements in $\Sigma_i^{n_i}$ and apply $C_i$'s decoding algorithm to each one to find an element in $\Sigma_i^{k_i}$. Then we map each element of $\Sigma_i^{k_i}$ to an element of $\Sigma_o$; the concatenation of all such elements forms a string in $\Sigma_o^{n_o}$. Applying the decoding algorithm for $C_o$ then returns the message as an element of $\Sigma_o^{k_o}$.

Figure 1 shows a graphical illustration of code concatenation. A message of size $k_1$ is expanded to a codeword of size $n_1$ by $C_o$. If the inner code has parameters $[n_i, k_i, d_i]_{q_i}$ and the outer code has parameters $[n_o, k_o, d_o]_{q_o}$, then the new code has parameters $[n_i \cdot n_o, k_i \cdot k_o, d_i \cdot d_o]_{q_i}$.

[2] In EE literature, what we describe is known as "serial concatenation." Other forms of concatenation, such as "parallel concatenation" are also studied.

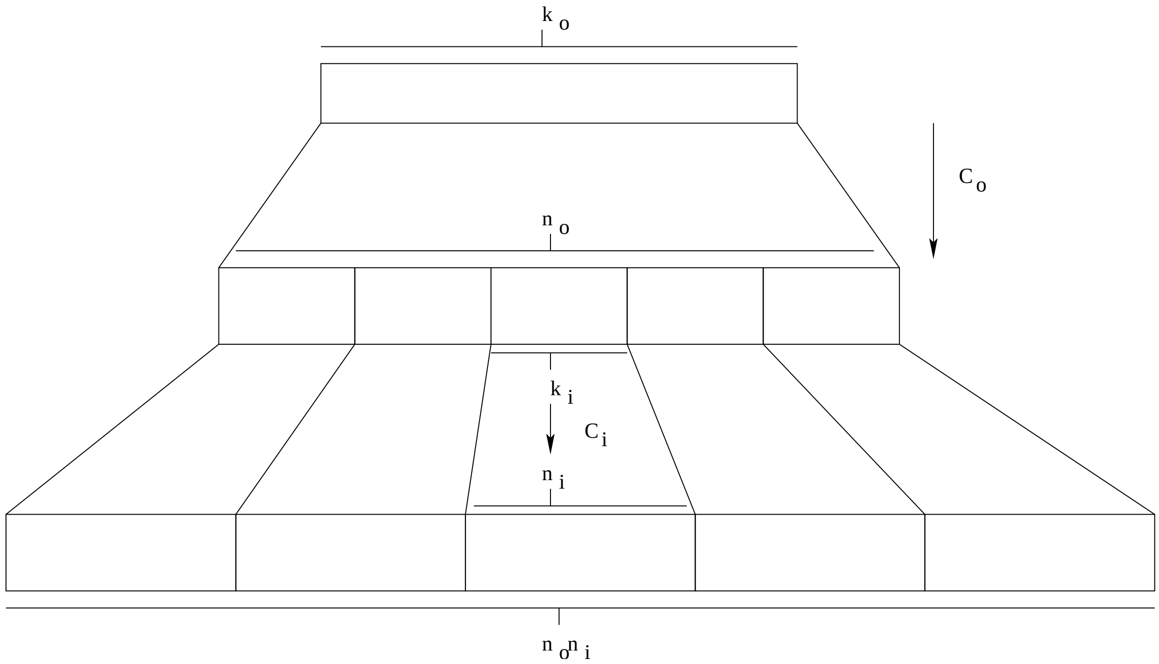


Figure 1: Concatenating $C_o$ and $C_i$.

As an example, let's consider concatenating a Reed-Solomon code with another Reed-Solomon code. Let $RS_1$ be a code with parameters $\left[n_1, \frac{n_1}{2}, \frac{n_1}{2}\right]_{n_1}$ and $RS_2$ be a code with parameters $\left[n_2, \frac{n_2}{2}, \frac{n_2}{2}\right]_{n_2}$. Now set $n = n_2^{\frac{n_2}{2}}$. The concatenated code then has parameters $\left[n_1, \frac{n}{4}, \frac{n}{4}\right]_{\frac{\log n}{\log \log n}}$. By repeating this process, we can make the alphabet size very small, roughly exponentially smaller each time.

### 2.3 Justesen Codes

Remember that we motivated concatenation as a tool for obtaining codes over smaller alphabets. Now that we have code concatenation, let's employ it to obtain an efficient binary code. Which codes should we concatenate? Reed-Solomon gives us a $[n, k, d]_q$ outer code, so now we need a good inner code.

The key idea here is that the alphabet size of the inner code is *exponentially small* compared to the alphabet size of the outer code. The Gilbert-Varshamov bound covered in last lecture tells us that most linear functions of a certain size are good error-correcting codes. This means we can apply brute force to find one such good code and use it for our inner code. The time to search all linear codes is polynomial in the size of the outer code.

It turns out we can dispense with actually searching for the optimal code. Justesen pointed out that if simply try a different linear code for each symbol, enough of them will be good enough for everything to work out[**?**]. That is, we perform the following operation for encoding:

1. Apply the Reed-Solomon encoding algorithm to the message. Divide the encoded

message into blocks $B_0, B_1, \ldots, B_k$ of $\lceil \log q \rceil$ bits each.

2. Enumerate all matrices $E_0, E_1, \ldots, E_k$ from $\{0,1\}^{\lceil \log q \rceil} \to \{0,1\}^{2\lceil \log q \rceil}$.

3. Output $E_0 \cdot B_0, E_1 \cdot B_1, \ldots, E_k \cdot B_k$.

For decoding the $i$th block of the message, we look up the appropriate encoding matrix $E_i$, attempt to invert it, and if successful multiply by the inverted matrix to decode. Some of these codes are "bad" and will fail to decode properly, but *most* will be good linear error-correcting codes. The Reed-Solomon decoding algorithm will then interpret the "bad" codes as noise and simply decode around them. The result is a binary error-correcting code with much better rate and minimum distance than we could have accomplished directly!

These concatenated codes are called *Justesen codes*. This is just a sketch of the idea. We'll return to Justesen codes in the next lecture and go through the analysis showing why they work, relating the proportion of good codes to the Gilbert-Varshamov bound, and derive their parameters. For now, these codes illustrate two key ideas:

1. The inner code is exponentially small in the size of the combined code. Therefore we can enumerate almost all inner codes or brute force the inner code.

2. The outer code's decoding algorithm can "make up for" errors decoding the inner code.

We will see these ideas again in the rest of the course, especially when we consider constructions of *list-decodable* and *list-recoverable* codes.