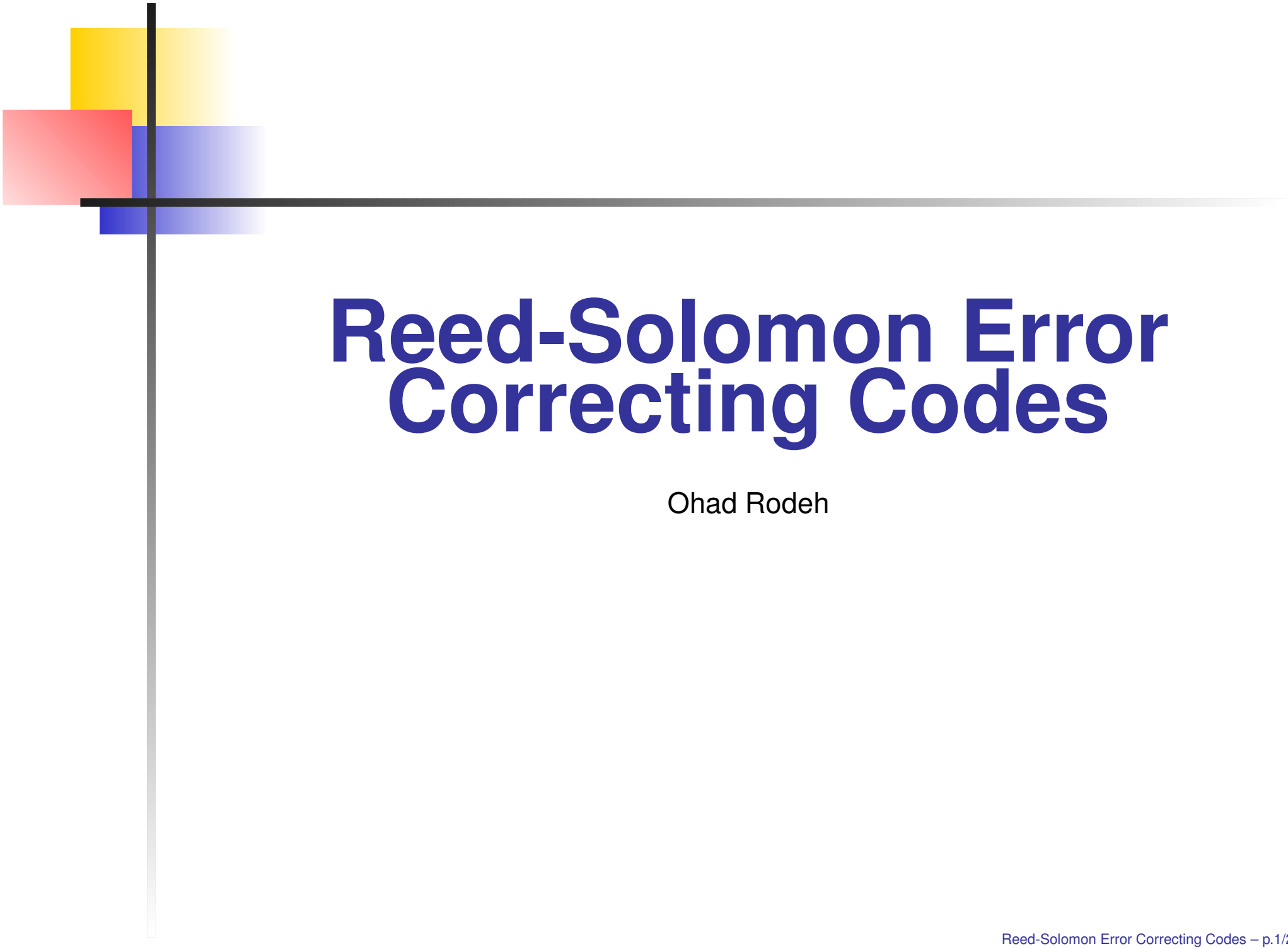
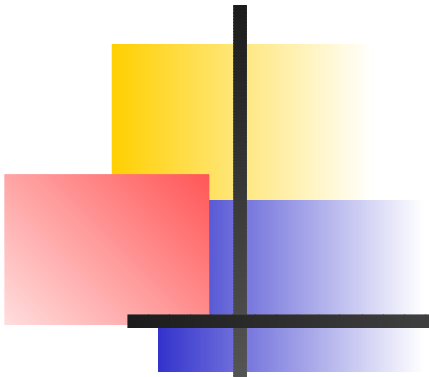


Reed-Solomon Error Correcting Codes

Ohad Rodeh



Introduction

- Reed-Solomon codes are used for error correction
- The code was invented in 1960 by Irving S. Reed and Gustave Solomon, at the MIT Lincoln Laboratory
- This lecture describes a particular application to storage
- It is based on: **A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like systems**, James S. Plank, University of Tennessee, 1996.

Problem definition

- n storage devices holding data: $D_1, \ldots, D_n$. Each holds k bytes.
- m storage devices holding checksums: $C_1, \ldots, C_m$. Each holds k bytes.
- The checksums are computed from the data.
- The goal: if any m devices fail, they can be reconstructed from the surviving devices.

Constructing the checksums

- The value of the checksum devices is computed using a function F

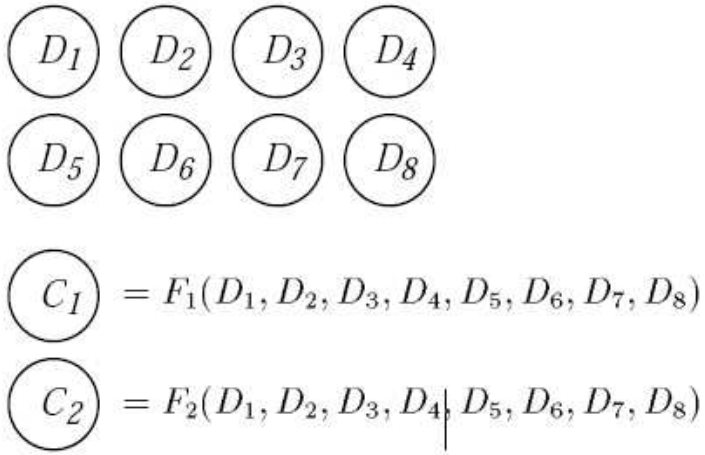


Figure 1: Providing two-site fault tolerance with two checksum devices

Words

- The RS-RAID scheme breaks the data into words of size w bits
- The coding scheme works on stripes whose width is w

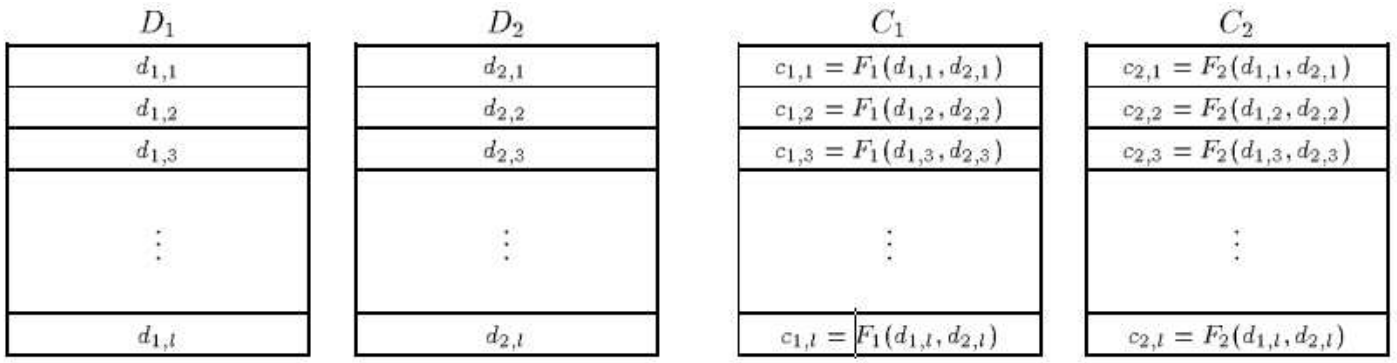


Figure 2: Breaking the storage devices into words ($n = 2$, $m = 2$, $l = \frac{8k}{w}$)

Simplifying the problem

- From here we focus on a single stripe
- The data words are $d_1, d_2, \ldots, d_n$
- The checksum words are $c_1, c_2, \ldots, c_m$

$$c_i = F(d_1, d_2, \ldots, d_n)$$

- If d_j changes to d'_j compute:

$$c'_i = G_{i,j}(d_j, d'_j, c_i)$$

- When data devices fail we re-compute the data from the available devices and then re-compute the failed checksum devices from the data devices

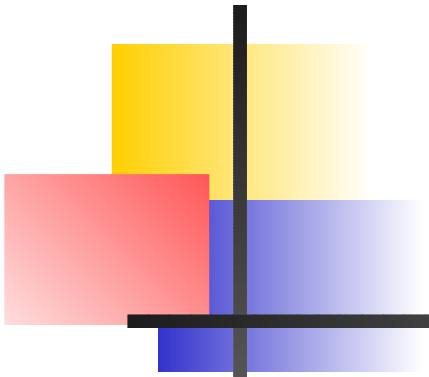
Example, RAID-4 as RS-RAID

- Set $m = 1$
- Describe $n + 1$ parity:

$$c_i = F(d_1, d_2, \ldots, d_n) = d_1 \oplus d_2 \oplus \cdots \oplus d_n$$

- If device j fails then:

$$d_j = d_1 \oplus \ldots d_{j-1} \oplus d_{j+1} \cdots \oplus d_n \oplus c_1$$



Restating the problem

- There are n data words $d_1, \ldots d_n$ all of size w
- We will
  1. Define functions F and G used to calculate the parity words $c_1, .., c_m$
  2. Describe how to recover after losing up to m devices
- Three parts to the solution:
  1. Using Vandermonde matrix to compute checksums
  2. Using Gaussian elimination to recover from failures
  3. Use of Galois fields to perform arithmetic

Vandermonde matrix

- Each checksum word c_i is a linear combination of the data words
- The matrix contains linearly independent rows

$$FD = C$$

$$\begin{bmatrix} f_{1,1} & f_{1,2} & \dots & f_{1,n} \\ f_{2,1} & f_{2,2} & \dots & f_{2,n} \\ \vdots & \vdots & & \vdots \\ f_{m,1} & f_{m,2} & \dots & f_{m,n} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} =$$

$$\begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & 2 & 3 & \dots & n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & 2^{m-1} & 3^{m-1} & \dots & n^{m-1} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}.$$

Changing a single word

- If data in device j changes from d_j to d'_j then

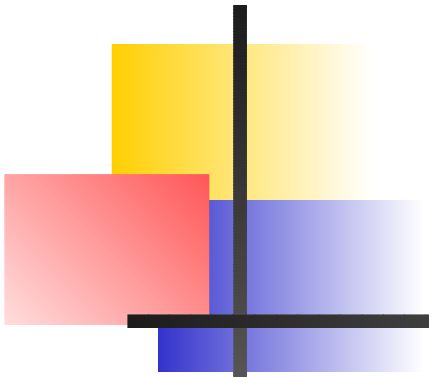
$$c'_i = G_{i,j}(d_j, d'_j, c_i) = c_i + f_{i,j}(d'_j - d_j)$$

- This makes changing a single word reasonably efficient

Recovering from failures

- In the matrix below, even after removing m rows, the matrix remains invertible

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ 1 & 1 & 1 & \dots & 1 \\ 1 & 2 & 3 & \dots & n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & 2^{m-1} & 3^{m-1} & \dots & n^{m-1} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \\ c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}$$



Galois fields

- We want to use bytes as our data and checksum elements. Therefore, we need a field with 256 elements.
- A field $GF(n)$ is a set of n elements closed under addition and multiplication
- Every element has an inverse for addition and multiplication

Primes

- If $n = p$ is a prime then $Z_p = \{0, \ldots, n-1\}$ is a field with addition and multiplication module p.
- For example, $GF(2) = \{0, 1\}$ with addition/multiplication modulo 2.
- If n is not a prime then addition/multiplication modulo n over the set $\{0, 1, .., n-1\}$ are not a field
  1. For example, if $n = 4$, and the elements are $\{0, 1, 2, 3\}$
  2. Element 2 does not have a multiplicative inverse

$GF(2^w)$

- Therefore, we cannot simply use elements $\{0, .., 2^n - 1\}$ with addition/multiplication module 2^n, we need to use Galois fields
- We use the Galois field $GF(2^w)$
- The elements of $GF(2^w)$ are polynomials with coefficients that are zero or one.
- Arithmetic in $GF(2^w)$ is like polynomial arithmetic module a primitive polynomial of degree w
- A primitive polynomial is one that is cannot be factored

$GF(4)$

- $GF(4)$ contains elements: $\{0, 1, x, x+1\}$
- The primitive polynomial is $q(x) = x^2 + x + 1$
- Arithmetic:

$$x + (x+1) = 1$$
$$x \times x = x^2 \mod (x^2 + x + 1) = x + 1$$
$$x \times (x+1) = x^2 + x \mod (x^2 + x + 1) = 1$$

Efficient arithmetic

- It is important to implement arithmetic efficiently over $GF(2^w)$
- Elements in $GF(2^n)$ are mapped onto the integers $0, \ldots, 2^n - 1$
- Mapping of $r(x)$ onto word of size w
- i-th bit is equal to the coefficient of x_i in $r(x)$
- Addition is performed with XOR
- How do we perform efficient multiplication? Polynomial multiplication is too slow.

Primitive polynomial

- A primitive polynomial $q(x)$ is one that cannot be factored
- If $q(x)$ is primitive then x generates all the elements in $GF(2^n)$

Generated Element of $GF(4)$	Polynomial Element of $GF(4)$	Binary Element b of $GF(4)$	Decimal Representation of b
0	0	00	0
x^0	1	01	1
x^1	x	10	2
x^2	$x+1$	11	3

Multiplication

- Multiplication (and division) is done using logarithms
- $a \times b = x^{log_x(a)+log_x(b)}$
- Build two tables
  1. gflog[] : maps an integer to its logarithm
  2. gfilog[] : maps an integer to its inverse logarithm
- In order to multiply two integers, a and b
  1. Compute their logarithms
  2. Add
  3. Compute the inverse logarithm
- For division: subtract instead of add

$GF(16)$

- Example tables for $GF(16)$
- Example computations:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
gflog[i]	$-\infty$	0	1	4	2	8	5	10	3	14	9	7	6	13	11	12
gfilog[i]	1	2	4	8	3	6	12	11	5	10	7	14	15	13	9	1

Table 1: Logarithm tables for $GF(16)$

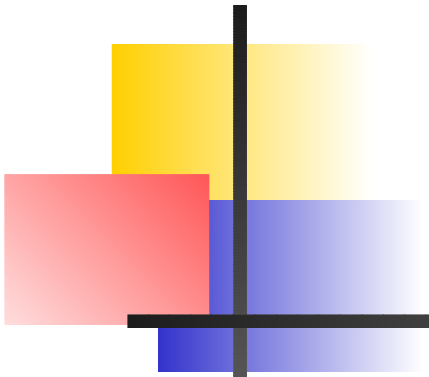
For example, in $GF(16)$:

$$
\begin{aligned}
3 * 7 &= \texttt{gfilog[gflog[3]+gflog[7]]} = \texttt{gfilog[4+10]} = \texttt{gfilog[14]} = 9 \\
13 * 10 &= \texttt{gfilog[gflog[13]+gflog[10]]} = \texttt{gfilog[13+9]} = \texttt{gfilog[7]} = 11 \\
13 \div 10 &= \texttt{gfilog[gflog[13]-gflog[10]]} = \texttt{gfilog[13-9]} = \texttt{gfilog[4]} = 3 \\
3 \div 7 &= \texttt{gfilog[gflog[3]-gflog[7]]} = \texttt{gfilog[4-10]} = \texttt{gfilog[9]} = 14
\end{aligned}
$$

$GF(16)$ cont.

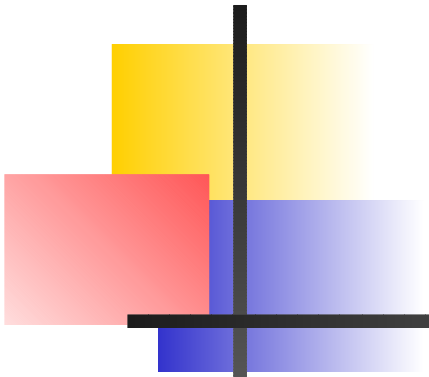
Generated Element	Polynomial Element	Binary Element	Decimal Element
0	0	0000	0
x^0	1	0001	1
x^1	x	0010	2
x^2	x^2	0100	4
x^3	x^3	1000	8
x^4	$x+1$	0011	3
x^5	x^2+x	0110	6
x^6	x^3+x^2	1100	12
x^7	x^3+x+1	1011	11
x^8	x^2+1	0101	5
x^9	x^3+x	1010	10
x^{10}	x^2+x+1	0111	7
x^{11}	x^3+x^2+x	1110	14
x^{12}	x^3+x^2+x+1	1111	15
x^{13}	x^3+x^2+1	1101	13
x^{14}	x^3+1	1001	9
x^{15}	1	0001	1

Table 2: Enumeration of the elements of $GF(16)$



Summary

- Choose a convenient value for w, $w = 8$ is a good choice
- Setup the tables gflog and gfilog
- Setup the Vandermonde matrix, arithmetic is over $GF(2^w)$
- Use F to maintain the checksums
- If any device fails, use F to reconstruct the data



Conclusions

- We presented Reed-Solomon error correction codes
- Advantages:
  1. Conceptually simple
  2. Can work with any n and m
- Disadvantage: Computationally more expensive than XOR based codes

Reed–Solomon error correction

From Wikipedia, the free encyclopedia

Reed–Solomon error correction is an error-correcting code that works by oversampling a polynomial constructed from the data. The polynomial is evaluated at several points, and these values are sent or recorded. Sampling the polynomial more often than is necessary makes the polynomial over-determined. As long as it receives "many" of the points correctly, the receiver can recover the original polynomial even in the presence of a "few" bad points.

Reed–Solomon codes are used in a wide variety of commercial applications, most prominently in CDs, DVDs and Blu-ray Discs, in data transmission technologies such as DSL & WiMAX, in broadcast systems such as DVB and ATSC, and in computer applications such as RAID 6 systems.

Contents

Overview

Reed–Solomon codes are block codes. This means that a fixed block of input data is processed into a fixed block of output data. In the case of the most commonly used R–S code (255, 223) – 223 Reed–Solomon input symbols (each eight bits long) are encoded into 255 output symbols.

- Most R–S error-correcting code schemes are systematic. This means that some portion of the output codeword contains the input data in its original form.
- A Reed–Solomon symbol size of eight bits forces the longest codeword length to be 255 symbols.
- The standard (255, 223) Reed–Solomon code is capable of correcting up to 16 Reed–Solomon symbol errors in each codeword. Since each symbol is actually eight bits, this means that the code can correct up to 16 short bursts of error.

The Reed–Solomon code, like the convolutional code, is a transparent code. This means that if the channel symbols have been inverted somewhere along the line, the decoders will still operate. The result will be the complement of the original data. However, the Reed–Solomon code loses its transparency when the code is shortened (see below). The "missing" bits in a shortened code need to be filled by either zeros or ones, depending on whether the data is complemented or not. (To put it another way, if the symbols are inverted, then the zero fill needs to be inverted to a ones fill.) For this reason it is mandatory that the sense of the data (i.e., true or complemented) be resolved before Reed–Solomon decoding.

Definition

Overview

The key idea behind a Reed–Solomon code is that the data encoded is first visualized as a Gegenbauer polynomial. The code relies on a theorem from algebra that states that any k distinct points *uniquely* determine a univariate polynomial of degree, at most, $k - 1$.

The sender determines a degree $k - 1$ polynomial, over a finite field, that represents the k data points. The polynomial is then "encoded" by its evaluation at various points, and these values are what is actually sent. During transmission, some of these values may become corrupted. Therefore, more than k points are actually sent. As long as sufficient values are received correctly, the receiver can deduce what the original polynomial was, and hence decode the original data.

In the same sense that one can correct a curve by interpolating past a gap, a Reed–Solomon code can bridge a series of errors in a block of data to recover the coefficients of the polynomial that drew the original curve.

Mathematical formulation

Given a finite field F and polynomial ring $F[x]$, let n and k be chosen such that $1 \leq k \leq n \leq |F|$. Pick n distinct elements of F, denoted $\{x_1, x_2, \ldots, x_n\}$. Then, the codebook $\mathbf{C}$ is created from the tuplets of values obtained by evaluating every polynomial (over F) of degree less than k at each x_i; that is,

$$\mathbf{C} = \{(f(x_1), f(x_2), \ldots, f(x_n)) \,|\, f \in F[x], \deg(f) < k\}\,.$$

C is a $[n,k,n-k+1]$ code; in other words, it is a linear code of length n (over F) with dimension k and minimum Hamming distance $n-k+1$.

A *Reed–Solomon code* is a code of the above form, subject to the additional requirement that the set $\{x_1, x_2, \ldots, x_n\}$ must be the set of *all* non-zero elements of the field F (and therefore, $n = |F| - 1$).

Remarks

For practical uses of Reed–Solomon codes, it is common to use a finite field F with 2^m elements. In this case, each symbol can be represented as an m-bit value. The sender sends the data points as encoded blocks, and the number of symbols in the encoded block is $n = 2^m - 1$. Thus a Reed–Solomon code operating on 8-bit symbols has $n = 2^8 - 1 = 255$ symbols per block. (This is a very popular value because of the prevalence of byte-oriented computer systems.) The number k, with $k < n$, of *data* symbols in the block is a design parameter. A commonly used code encodes $k = 223$ eight-bit data symbols plus 32 eight-bit parity symbols in an $n = 255$-symbol block; this is denoted as a $(n,k) = (255,223)$ code, and is capable of correcting up to 16 symbol errors per block.

The set $\{x_1,\ldots,x_n\}$ of non-zero elements of a finite field can be written as $\{1,\alpha,\alpha^2,\ldots,\alpha^{n-1}\}$, where α is a primitive nth root of unity. It is customary to encode the values of a Reed–Solomon code in this order. Since $\alpha^n = 1$, and since for every polynomial $p(x)$, the function $p(\alpha x)$ is also a polynomial of the same degree, it then follows that a Reed–Solomon code is cyclic.

Reed–Solomon codes as BCH codes

Reed–Solomon codes are a special case of a larger class of codes called BCH codes. An efficient error correction algorithm for BCH codes (and therefore Reed–Solomon codes) was discovered by Berlekamp in 1968.

To see that Reed–Solomon codes are special BCH codes, it is useful to give the following alternate definition of Reed–Solomon codes.[1]

Given a finite field F of size q, let $n = q - 1$ and let α be a primitive nth root of unity in F. Also let $1 \le k \le n$ be given. The *Reed–Solomon code* for these parameters has code word $(f_0,f_1,\ldots,f_{n-1})$ if and only if $\alpha,\alpha^2,\ldots,\alpha^{n-k}$ are roots of the polynomial

$$p(x) = f_0 + f_1 x + \ldots + f_{n-1} x^{n-1}.$$

With this definition, it is immediately seen that a Reed–Solomon code is a polynomial code, and in particular a BCH code. The generator polynomial $g(x)$ is the minimal polynomial with roots $\alpha,\alpha^2,...,\alpha^{n-k}$, and the code words are exactly the polynomials that are divisible by $g(x)$.

Equivalence of the two formulations

At first sight, the above two definitions of Reed–Solomon codes seem very different. In the first definition, code words are *values* of polynomials, whereas in the second, they are *coefficients*. Moreover, the polynomials in the first definition are required to be of small degree, whereas those in the second definition are required to have specific roots.

The equivalence of the two definitions is proved using the discrete Fourier transform. This transform, which exists in all finite fields as well as the complex numbers, establishes a duality between the coefficients of polynomials and their values. This duality can be approximately summarized as follows: Let $p(x)$ and $q(x)$ be two polynomials of degree less than n. If the *values* of $p(x)$ are the *coefficients* of $q(x)$, then (up to a scalar factor and reordering), the *values* of $q(x)$ are the *coefficients* of $p(x)$. For this to make sense, the values must be taken at locations $x = \alpha^i$, for $i = 0, \ldots, n-1$, where α is a primitive nth root of unity.

To be more precise, let

$$p(x) = v_0 + v_1 x + v_2 x^2 + \ldots + v_{n-1} x^{n-1},$$
$$q(x) = f_0 + f_1 x + f_2 x^2 + \ldots + f_{n-1} x^{n-1}$$

and assume $p(x)$ and $q(x)$ are related by the discrete Fourier transform. Then the coefficients and values of $p(x)$ and $q(x)$ are related as follows: for all $i = 0, \ldots, n-1, f_i = p(\alpha^i)$ and $v_i = \frac{1}{n} q(\alpha^{n-i})$.

Using these facts, we have: $(f_0, \ldots, f_{n-1})$ is a code word of the Reed–Solomon code according to the first definition

- if and only if $p(x)$ is of degree less than k (because $f_0, \ldots, f_{n-1}$ are the values of $p(x)$),
- if and only if $v_i = 0$ for $i = k,...,n-1$,
- if and only if $q(\alpha^i) = 0$ for $i = 1,...,n-k$ (because $q(\alpha^i) = nv_{n-i}$),
- if and only if $(f_0, \ldots, f_{n-1})$ is a code word of the Reed–Solomon code according to the second definition.

This shows that the two definitions are equivalent.

Properties of Reed–Solomon codes

The error-correcting ability of any Reed–Solomon code is determined by $n - k$, the measure of redundancy in the block. If the locations of the errored symbols are not known in advance, then a Reed–Solomon code can correct up to $(n - k) / 2$ erroneous symbols, i.e., it can correct half as many errors as there are redundant symbols added to the block. Sometimes error locations are known in advance (e.g., "side information" in demodulator signal-to-noise ratios)—these are called **erasures**. A Reed–Solomon code (like any MDS code) is able to correct twice as many erasures as errors, and any combination of errors and erasures can be corrected as long as the relation $2E + S \leq n - k$ is satisfied, where E is the number of errors and S is the number of erasures in the block.

The properties of Reed–Solomon codes make them especially well-suited to applications where errors occur in bursts. This is because it does not matter to the code how many bits in a symbol are in error—if multiple bits in a symbol are corrupted it only counts as a single error. Conversely, if a data stream is not characterized by error bursts or drop-outs but by random single bit errors, a Reed–Solomon code is usually a poor choice.

Designers are not required to use the "natural" sizes of Reed–Solomon code blocks. A technique known as "shortening" can produce a smaller code of any desired size from a larger code. For example, the widely used (255,223) code can be converted to a (160,128) code by padding the unused portion of the block (usually the beginning) with 95 binary zeroes and not transmitting them. At the decoder, the same portion of the block is loaded locally with binary zeroes. The Delsarte-Goethals-Seidel[2] theorem illustrates an example of an application of shortened Reed–Solomon codes.

In 1999 Madhu Sudan and Venkatesan Guruswami at MIT, published "Improved Decoding of Reed–Solomon and Algebraic-Geometry Codes" introducing an algorithm that allowed for the correction of errors beyond half the minimum distance of the code. It applies to Reed–Solomon codes and more generally to algebraic geometry codes. This algorithm produces a list of codewords (it is a list-decoding algorithm) and is based on interpolation and factorization of polynomials over $GF(2^m)$ and its extensions.

History

The code was invented in 1960 by Irving S. Reed and Gustave Solomon, who were then members of MIT Lincoln Laboratory. Their seminal article was entitled "Polynomial Codes over Certain Finite Fields." When it was written, digital technology was not advanced enough to implement the concept. The first application, in 1982, of RS codes in mass-produced products was the compact disc, where two interleaved RS codes are used. An efficient decoding algorithm for large-distance RS codes was developed by Elwyn Berlekamp and James Massey in 1969. Today RS codes are used in hard disk drive, DVD, telecommunication, and digital broadcast protocols.

Applications

Data storage

Reed–Solomon coding is very widely used in mass storage systems to correct the burst errors associated with media defects.

Reed–Solomon coding is a key component of the compact disc. It was the first use of strong error correction coding in a mass-produced consumer product, and DAT and DVD use similar schemes. In the CD, two layers of Reed–Solomon coding separated by a 28-way convolutional interleaver yields a scheme called Cross-Interleaved Reed Solomon Coding (CIRC). The first element of a CIRC decoder is a relatively weak inner (32,28) Reed–Solomon code, shortened from a (255,251) code with 8-bit symbols. This code can correct up to 2 byte errors per 32-byte block. More importantly, it flags as erasures any uncorrectable blocks, i.e., blocks with more than 2 byte errors. The decoded 28-byte blocks, with erasure indications, are then spread by the deinterleaver to different blocks of the (28,24) outer code. Thanks to the deinterleaving, an erased 28-byte block from the inner code becomes a single erased byte in each of 28 outer code blocks. The outer code easily corrects this, since it can handle up to 4 such erasures per block.

The result is a CIRC that can completely correct error bursts up to 4000 bits, or about 2.5 mm on the disc surface. This code is so strong that most CD playback errors are almost certainly caused by tracking errors that cause the laser to jump track, not by uncorrectable error bursts.[3]

Another product which incorporates Reed–Solomon coding is Nintendo's e-Reader. This is a video-game delivery system which uses a two-dimensional "barcode" printed on trading cards. The cards are scanned using a device which attaches to Nintendo's Game Boy Advance game system.

Reed–Solomon error correction is also used in parchive files which are commonly posted accompanying multimedia files on USENET. The Distributed online storage service Wuala also makes use of Reed–Solomon when breaking up files.

Data transmission

Specialized forms of Reed–Solomon codes specifically Cauchy-RS and Vandermonde-RS can be used to overcome the unreliable nature of data transmission over erasure channels. The encoding process assumes a code of RS(N,K) which results in N codewords of length N symbols each storing K symbols of data, being generated, that are then sent over an erasure channel.

Any combination of K codewords received at the other end is enough to reconstruct all of the N codewords. The code rate is generally set to 1/2 unless the channel's erasure likelihood can be adequately modelled and is seen to be less. In conclusion N is usually 2K, meaning that at least half of all the codewords sent must be received in order to reconstruct all of the codewords sent.

Reed–Solomon codes are also used in xDSL systems and CCSDS's Space Communications Protocol Specifications as a form of Forward Error Correction.

Mail encoding

Paper bar codes such as PostBar, MaxiCode, Datamatrix and QR Code use Reed–Solomon error correction to allow correct reading even if a portion of the bar code is damaged.

Satellite transmission

One significant application of Reed–Solomon coding was to encode the digital pictures sent back by the Voyager space probe.

Voyager introduced Reed–Solomon coding concatenated with convolutional codes, a practice that has since become very widespread in deep space and satellite (e.g., direct digital broadcasting) communications.

Viterbi decoders tend to produce errors in short bursts. Correcting these burst errors is a job best done by short or simplified Reed–Solomon codes.

Modern versions of concatenated Reed–Solomon/Viterbi-decoded convolutional coding were and are used on the Mars Pathfinder, Galileo, Mars Exploration Rover and Cassini missions, where they perform within about 1–1.5 dB of the ultimate limit imposed by the Shannon capacity.

These concatenated codes are now being replaced by more powerful turbo codes where the transmitted data does not need to be decoded immediately.

Sketch of the error correction algorithm

The following is a sketch of the main idea behind the error correction algorithm for Reed–Solomon codes.

By definition, a code word of a Reed–Solomon code is given by the sequence of values of a low-degree polynomial over a finite field. A key fact for the error correction algorithm is that the *values* and the *coefficients* of a polynomial are related by the discrete Fourier transform.

The purpose of a Fourier transform is to convert a signal from a time domain to a frequency domain or vice versa. In case of the Fourier transform over a finite field, the frequency domain signal corresponds to the coefficients of a polynomial, and the time domain signal correspond to the values of the same polynomial.

As shown in Figures 1 and 2, an isolated value in the frequency domain corresponds to a smooth wave in the time domain. The wavelength depends on the location of the isolated value.

Conversely, as shown in Figures 3 and 4, an isolated value in the time domain corresponds to a smooth wave in the frequency domain.

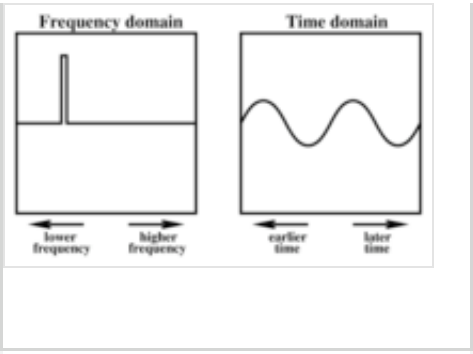


Figure 1

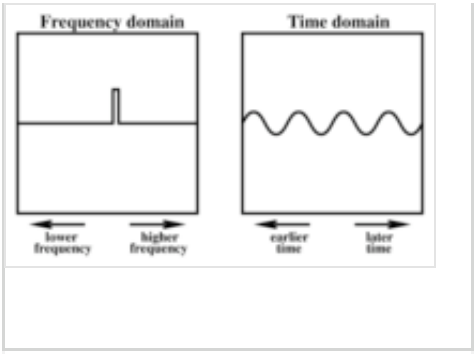


Figure 2

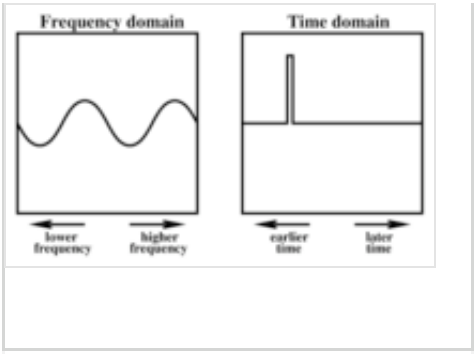


Figure 3

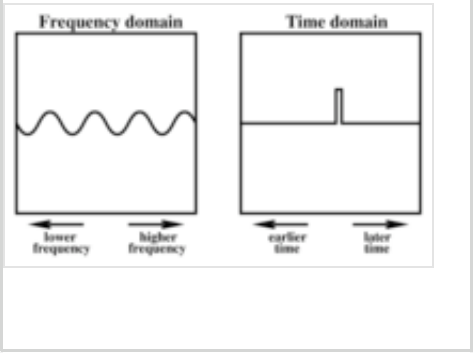


Figure 4

In a Reed–Solomon code, the frequency domain is divided into two regions as shown in Figure 5: a left (low-frequency) region of length k, and a right (high-frequency) region of length $n-k$. A data word is then embedded into the left region (corresponding to the k coefficients of a polynomial of degree at most $k-1$), while the right region is filled with zeros. The result is Fourier transformed into the time domain, yielding a code word that is composed only of low frequencies. In the absence of errors, a code word can be decoded by reverse Fourier transforming it back into the frequency domain.

Now consider a code word containing a single error, as shown in red in Figure 6. The effect of this error in the frequency domain is a smooth, single-frequency wave in the right region, called the *syndrome* of the error. The error location can be determined by determining the frequency of the syndrome signal.

Similarly, if two or more errors are introduced in the code word, the syndrome will be a signal composed of two or more frequencies, as shown in Figure 7. As long as it is possible to determine the frequencies of which the syndrome is composed, it is possible to determine the error locations. Notice that the error *locations* depend only on the *frequencies* of these waves, whereas the error *magnitudes* depend on their amplitudes and phase.

The problem of determining the error locations has therefore been reduced to the problem of finding, given a sequence of $n-k$ values, the smallest set of elementary waves into which these values can be decomposed. It is known from digital signal processing that this problem is equivalent to finding the roots of the minimal

polynomial of the sequence, or equivalently, of finding the shortest linear feedback shift register (LFSR) for the sequence. The latter problem can either be solved inefficiently by solving a system of linear equations, or more efficiently by the Berlekamp-Massey algorithm.

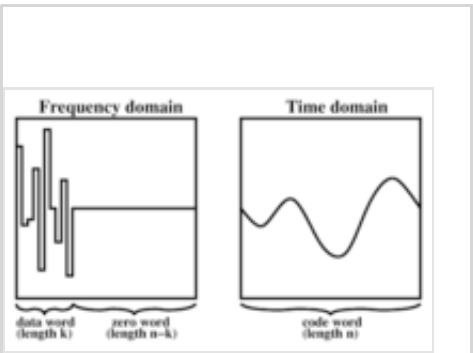


Figure 5

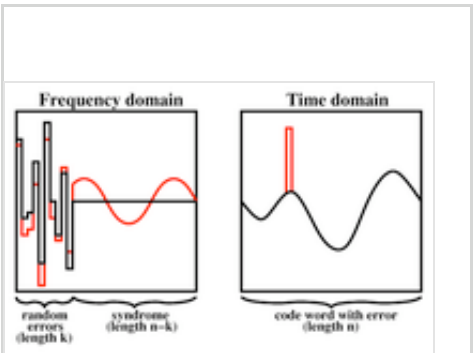


Figure 6

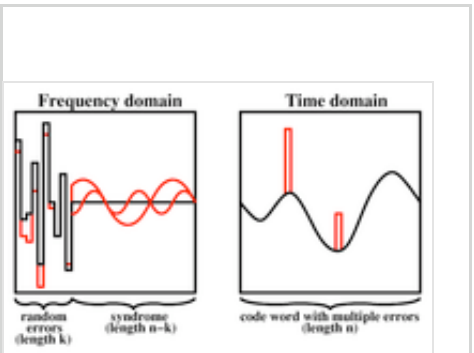


Figure 7

See also

- Forward error correction
- BCH code
- Low-density parity-check code
- Chien search
- Datamatrix
- Tornado codes
- Finite ring

References

1. ^ Lidl, Rudolf; Pilz, Günter (1999). *Applied Abstract Algebra* (2nd ed.). Wiley. p. 226.
2. ^ "Kissing Numbers, Sphere Packings, and Some Unexpected Proofs", Notices of the American Mathematical Society, Volume 51, Issue 8, 2004/09. Explains the Delsarte-Goethals-Seidel (http://www.ams.org/notices/200408/fea-pfender.pdf) theorem as used in the context of the error correcting code for compact disc.
3. ^ K.A.S. Immink, *Reed–Solomon Codes and the Compact Disc* in S.B. Wicker and V.K. Bhargava, Edrs, *Reed–Solomon Codes and Their Applications*, IEEE Press, 1994.

- F. J. MacWilliams and N. J. A. Sloane, The Theory of Error-Correcting Code, New York: North-Holland Publishing Company, 1977.
- Irving S. Reed and Xuemin Chen, Error-Control Coding for Data Networks", Boston: Kluwer Academic Publishers, 1999.
- MIT OpenCourseWare - Principles of Digital Communication II - Lecture 10 on Reed Solomon Codes (http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-451Spring-2005/LectureNotes/detail/embed10.htm)

External links

- Schifra Open Source C++ Reed–Solomon Codec (http://www.schifra.com)
- A collection of links to books, online articles and source code (http://www.radionetworkprocessor.com/reed-solomon.html)
- Henry Minsky's RSCode library, Reed–Solomon encoder/decoder (http://rscode.sourceforge.net/)
- A Tutorial on Reed–Solomon Coding for Fault-Tolerance in RAID-like Systems (http://www.cs.utk.edu/%7Eplank/plank/papers/SPE-9-97.html)
- A thesis on Algebraic soft-decoding of Reed–Solomon codes (http://sidewords.files.wordpress.com/2007/12/thesis.pdf) . It explains the basics as well.
- Matlab implementation of errors-and-erasures Reed-Solomon decoding (http://dept.ee.wits.ac.za/~versfeld/research_resources/sourcecode/Errors_And_Erasures_Test.zip)
- BBC R&D White Paper WHP031 (http://www.bbc.co.uk/rd/pubs/whp/whp031.shtml)

Retrieved from "http://en.wikipedia.org/wiki/Reed%E2%80%93Solomon_error_correction"
Categories: Error detection and correction

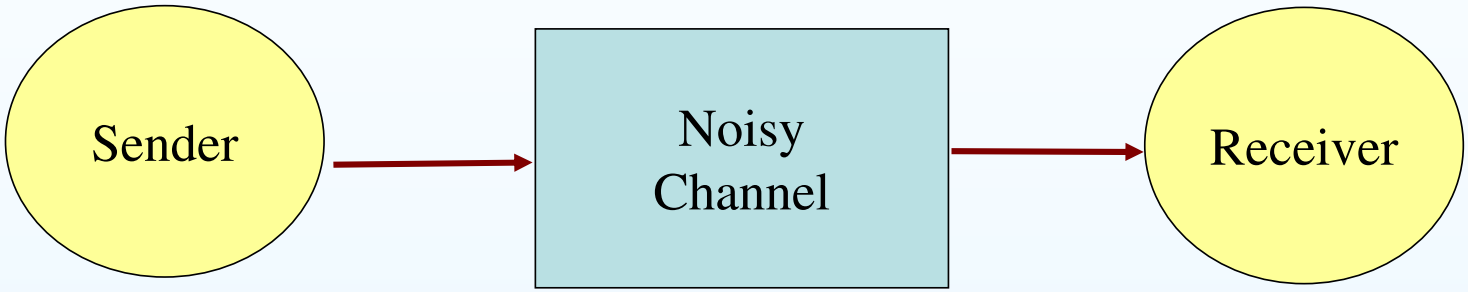
List Decoding of Reed Solomon Codes

Madhu Sudan

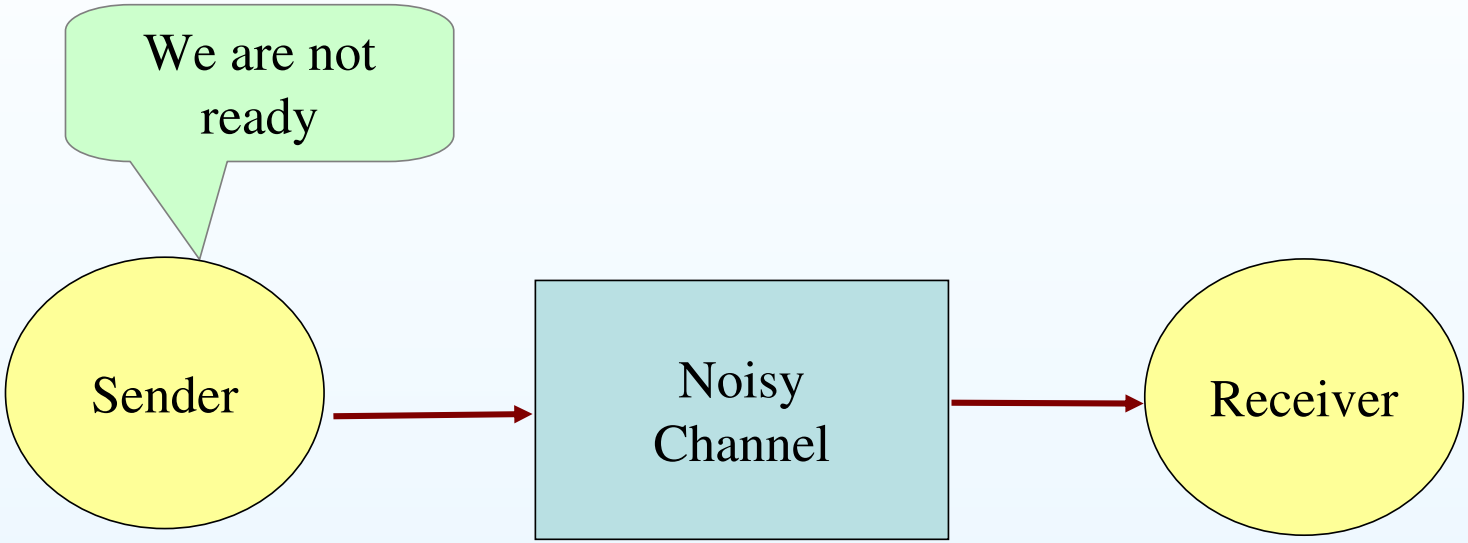
MIT CSAIL

Background: Reliable Transmission of Information

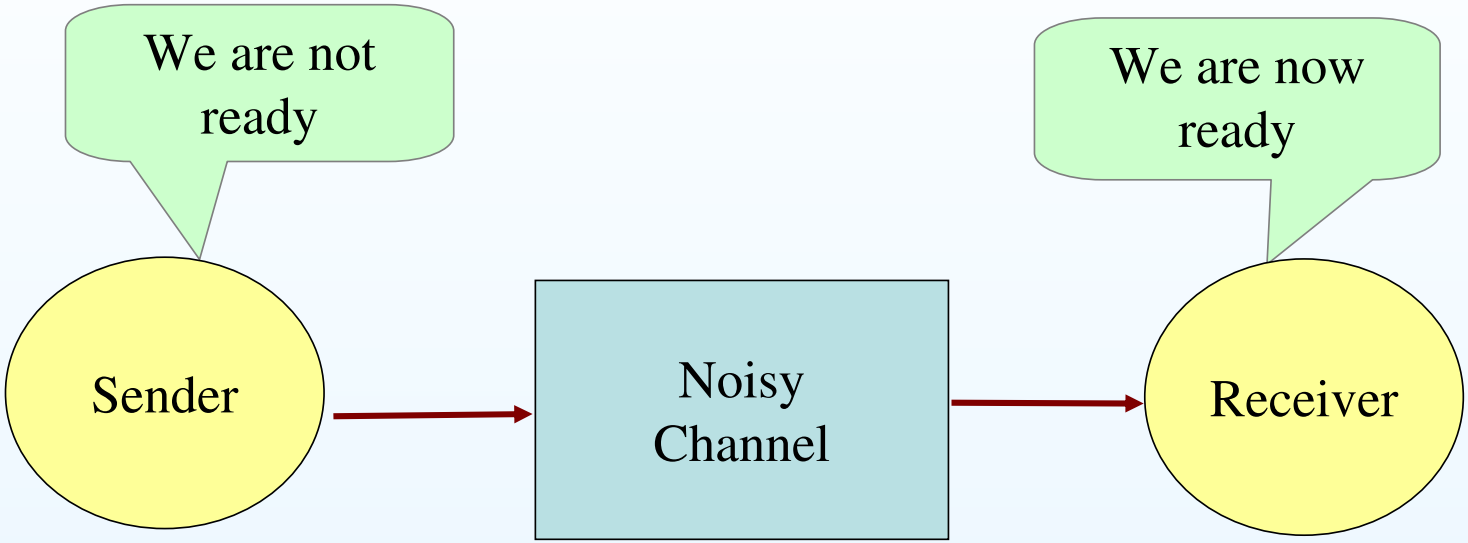
The Problem of Information Transmission



The Problem of Information Transmission



The Problem of Information Transmission



- When information is digital, reliability is critical.
- Need to understand errors, and correct them.

Shannon (1948)

- Model noise by probability distribution.
- Example: Binary symmetric channel (BSC)
  - Parameter $p \in [0, \frac{1}{2}]$.
  - Channel transmits bits.
  - With probability $1 - p$ bit transmitted faithfully, and with probability p bit flipped (independent of all other events).

Shannon's architecture

- Sender encodes k bits into n bits.
- Transmits n bit string on channel.
- Receiver decodes n bits into k bits.
- Rate of channel usage = k/n.

Shannon's theorem

- Every channel (in broad class) has a capacity s.t., transmitting at Rate below capacity is feasible and above capacity is infeasible.
- Example: Binary symmetric channel (p) has capacity $1 - H(p)$, where $H(p)$ is the binary entropy function.
  - $p = 0$ implies capacity $= 1$.
  - $p = \frac{1}{2}$ implies capacity $= 0$.
  - $p < \frac{1}{2}$ implies capacity > 0.
- Example: q-ary symmetric channel (p): On input $\sigma \in \mathbb{F}_q$ receiver receives (independently) σ', where
  - $\sigma' = \sigma$ w.p. $1 - p$.
  - σ' uniform over $\mathbb{F}_q - \{\sigma\}$ w.p. p.

  Capacity positive if $p < 1 - 1/q$.

Constructive versions

- Shannon's theory was non-constructive. Decoding takes exponential time.
- [Elias '55] gave polytime algorithms to achieve positive rate on every channel of positive capacity.
- [Forney '66] achieved any rate $<$ capacity with polynomial time algorithms (and exponentially small error).
- Modern results (following [Spielman '96]) lead to linear time algorithms.

Hamming (1950)

- Modelled errors adversarially.
- Focussed on image of encoding function (the "Code").
- Introduced metric (Hamming distance) on range of encoding function. $d(x, y) = \#$ coordinates such that $x_i \neq y_i$.
- Noticed that for adversarial error (and guaranteed error recovery), distance of Code is important.

$$\Delta(C) = \min_{x,y \in C} \{d(x, y)\}.$$

- Code of distance d corrects $(d-1)/2$ errors.

Contrast between Shannon & Hamming

Contrast between Shannon & Hamming

[Sha48] : $\mathcal{C}$ probabilistic.

E.g., flips each bit independently w.p. p.

✔ Tightly analyzed for many cases e.g., q-SC(p).

✗ Channel may be too weak to capture some scenarios.

✗ Need very accurate channel model.

Contrast between Shannon & Hamming

[Sha48] : $\mathcal{C}$ probabilistic.

✔ Corrects many errors. ✗ Channel restricted.

Contrast between Shannon & Hamming

[Sha48] : $\mathcal{C}$ probabilistic.

✔ Corrects many errors. ✗ Channel restricted.

[Ham50] : $\mathcal{C}$ flips bits *adversarially*

✔ Safer model, "good" codes known

✗ Too pessimistic: Can only decode if $p < 1/2$ for any alphabet.

Contrast between Shannon & Hamming

[Sha48] : $\mathcal{C}$ probabilistic.

✔ Corrects many errors. ✗ Channel restricted.

[Ham50] : $\mathcal{C}$ flips bits *adversarially*

✗ Fewer errors. ✔ More general errors.

Contrast between Shannon & Hamming

[Sha48] : $\mathcal{C}$ probabilistic.

✔ Corrects many errors. ✗ Channel restricted.

[Ham50] : $\mathcal{C}$ flips bits *adversarially*

✗ Fewer errors. ✔ More general errors.

- Which model is correct? Depends on application.
  - Crudely: Small $q \Rightarrow$ Shannon. Large $q \Rightarrow$ Hamming.
- Today: New Models of error-correction + algorithms.
  - List-decoding: Relaxed notion of decoding.

Contrast between Shannon & Hamming

[Sha48] : $\mathcal{C}$ probabilistic.

✔ Corrects many errors. ✗ Channel restricted.

[Ham50] : $\mathcal{C}$ flips bits *adversarially*

✗ Fewer errors. ✔ More general errors.

- Which model is correct? Depends on application.
  - Crudely: Small $q \Rightarrow$ Shannon. Large $q \Rightarrow$ Hamming.
- Today: New Models of error-correction + algorithms.
  - List-decoding: Relaxed notion of decoding.

    ✔ More errors ✔ Strong (enough) errors.

Reed-Solomon Codes

Motivation: [Singleton] Bound

- Suppose $C \subseteq \mathbb{F}_q^n$ has q^k codewords. How large can its distance be?

Motivation: [Singleton] Bound

- Suppose $C \subseteq \mathbb{F}_q^n$ has q^k codewords. How large can its distance be?
- Bound: $\Delta(C) \leq n - k + 1$.

Motivation: [Singleton] Bound

- Suppose $C \subseteq \mathbb{F}_q^n$ has q^k codewords. How large can its distance be?
- Bound: $\Delta(C) \leq n - k + 1$.
- Proof:
  - Project code to first $k - 1$ coordinates.
  - By Pigeonhole Principle, two codewords collide.
  - These two codewords thus disagree in at most $n - k + 1$ coordinates.

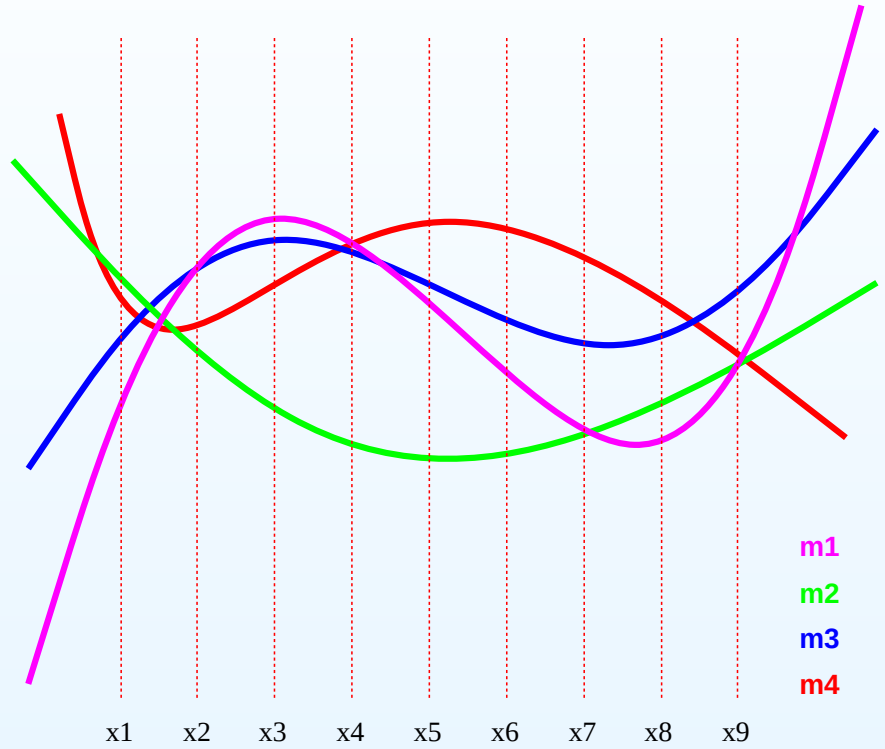
Motivation: [Singleton] Bound

- Suppose $C \subseteq \mathbb{F}_q^n$ has q^k codewords. How large can its distance be?
- Bound: $\Delta(C) \leq n - k + 1$.
- Proof:
  - Project code to first $k - 1$ coordinates.
  - By Pigeonhole Principle, two codewords collide.
  - These two codewords thus disagree in at most $n - k + 1$ coordinates.
- Surely we can do better?

Motivation: [Singleton] Bound

- Suppose $C \subseteq \mathbb{F}_q^n$ has q^k codewords. How large can its distance be?
- Bound: $\Delta(C) \leq n - k + 1$.
- Proof:
  - Project code to first $k - 1$ coordinates.
  - By Pigeonhole Principle, two codewords collide.
  - These two codewords thus disagree in at most $n - k + 1$ coordinates.
- Surely we can do better?
- Actually - No! [Reed-Solomon] Codes match this bound!

Reed-Solomon Codes



- Messages $\equiv$ Polynomial.
- Encoding $\equiv$ Evaluation at $x_1, \ldots, x_n$.
- $n >$ Degree: Injective
- $n \gg$ Degree: Redundant

Reed-Solomon Codes (formally)

- Let $\mathbb{F}_q$ be a finite field.
- Code specified by $k, n, \alpha_1, \ldots, \alpha_n \in \mathbb{F}_q$.
- Message: $\langle c_0, \ldots, c_k \rangle \in \mathbb{F}_q^{k+1}$ coefficients of degree k polynomial $p(x) = c_0 + c_1 x + \cdots c_k x^k$.
- Encoding: $p \mapsto \langle p(\alpha_1), \ldots, p(\alpha_n) \rangle$. ($k+1$ letters to n letters.)
- Degree k poly has at most k roots $\Leftrightarrow$ Distance $d = n - k$.
- These are the Reed-Solomon codes.
  Match [Singleton] bound!
  Commonly used (CDs, DVDs etc.).

List-Decoding of Reed-Solomon Codes

Reed-Solomon Decoding

Restatement of the problem:

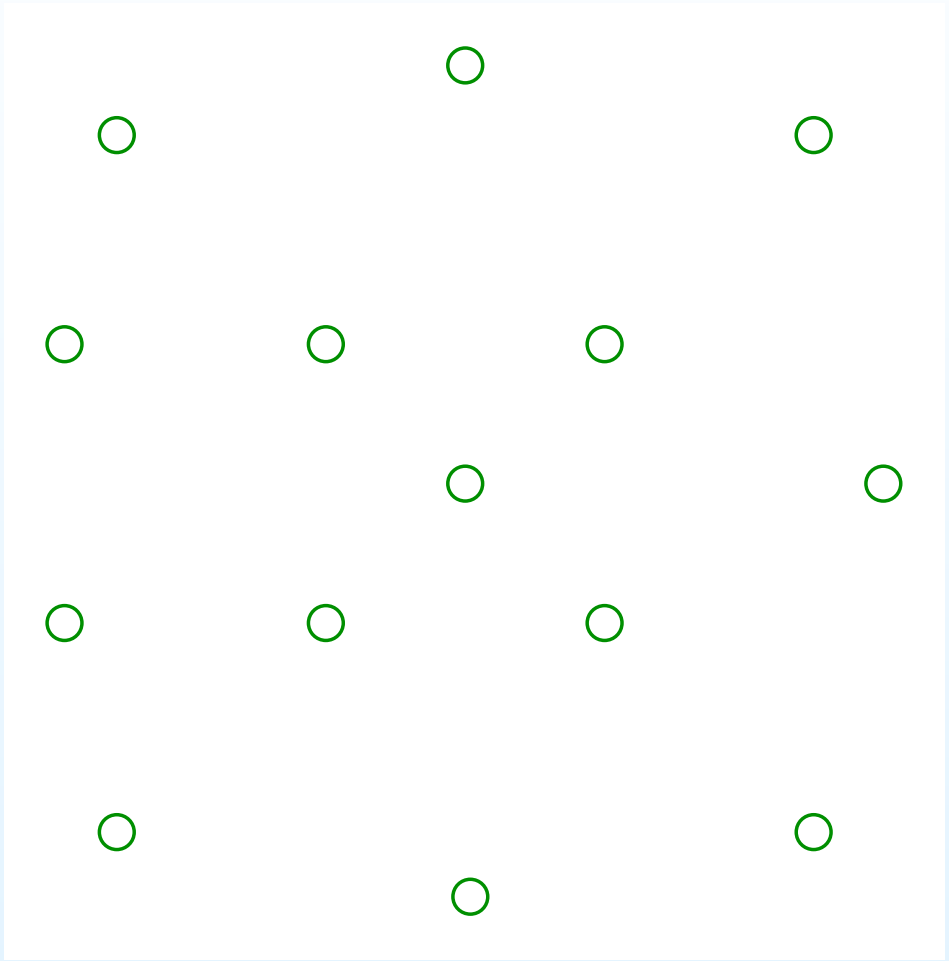
- Input: n points $(\alpha_i, y_i) \in \mathbb{F}_q^2$; agreement parameter t
- Output: **All** degree k polynomials $p(x)$ s.t. $p(\alpha_i) = y_i$ for at least t values of i.

We use $k = 1$ for illustration.

- i.e. want *all* "lines" $(y - ax - b = 0)$ that pass through $\geq t$ out of n points.

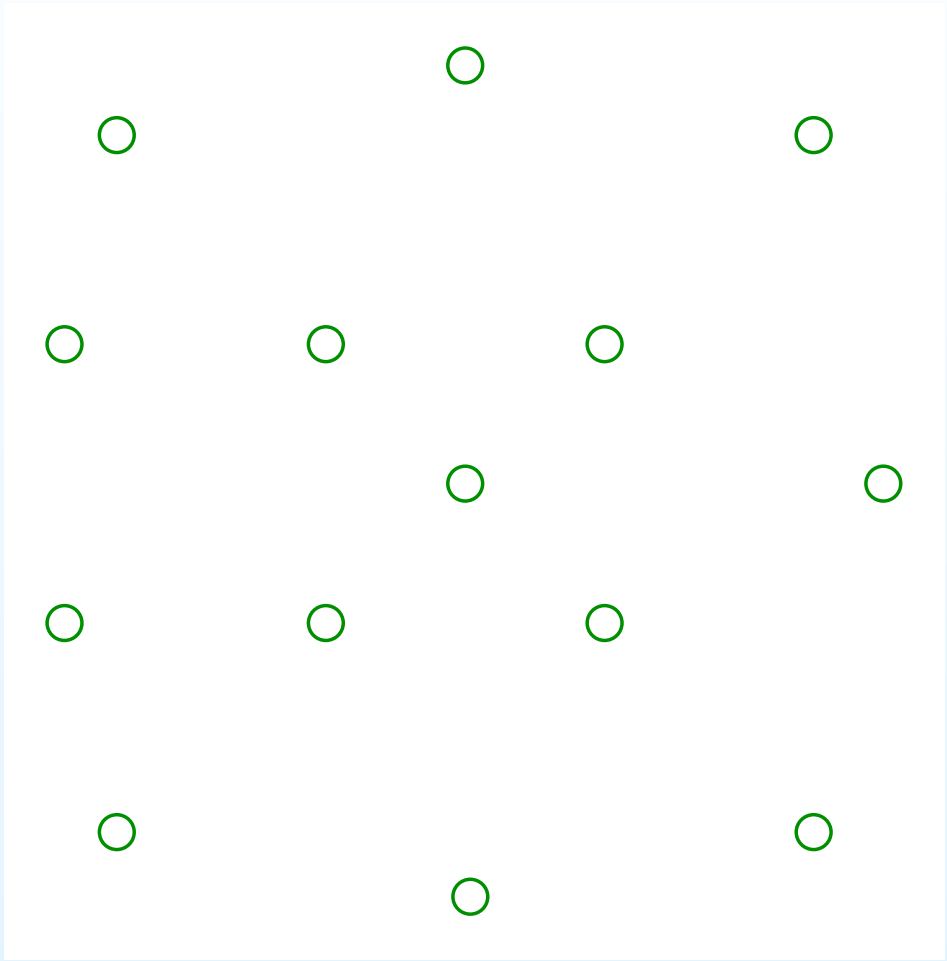
Algorithm Description [S. '96]

$n = 14$ points; Want all *lines* through at least **5** points.



Algorithm Description [S. '96]

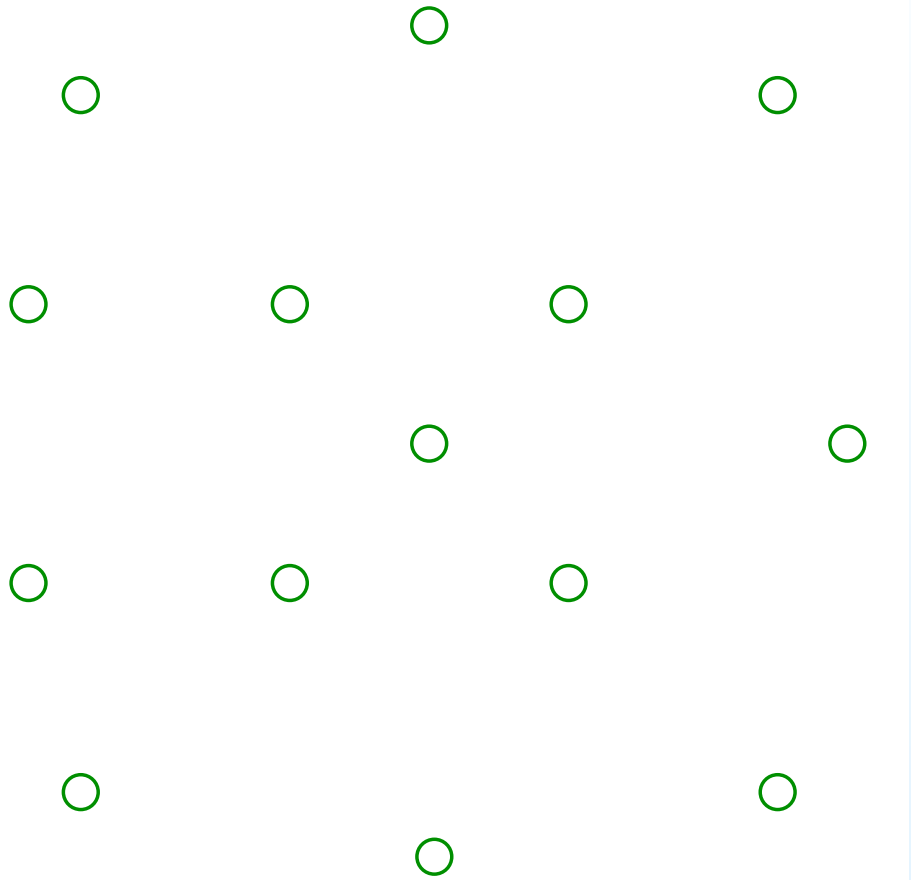
$n = 14$ points; Want all *lines* through at least **5** points.



Find deg. 4 poly. $Q(x, y) \not\equiv 0$
s.t. $Q(\alpha_i, y_i) = 0$ for all points.

Algorithm Description [S. '96]

$n = 14$ points; Want all *lines* through at least **5** points.



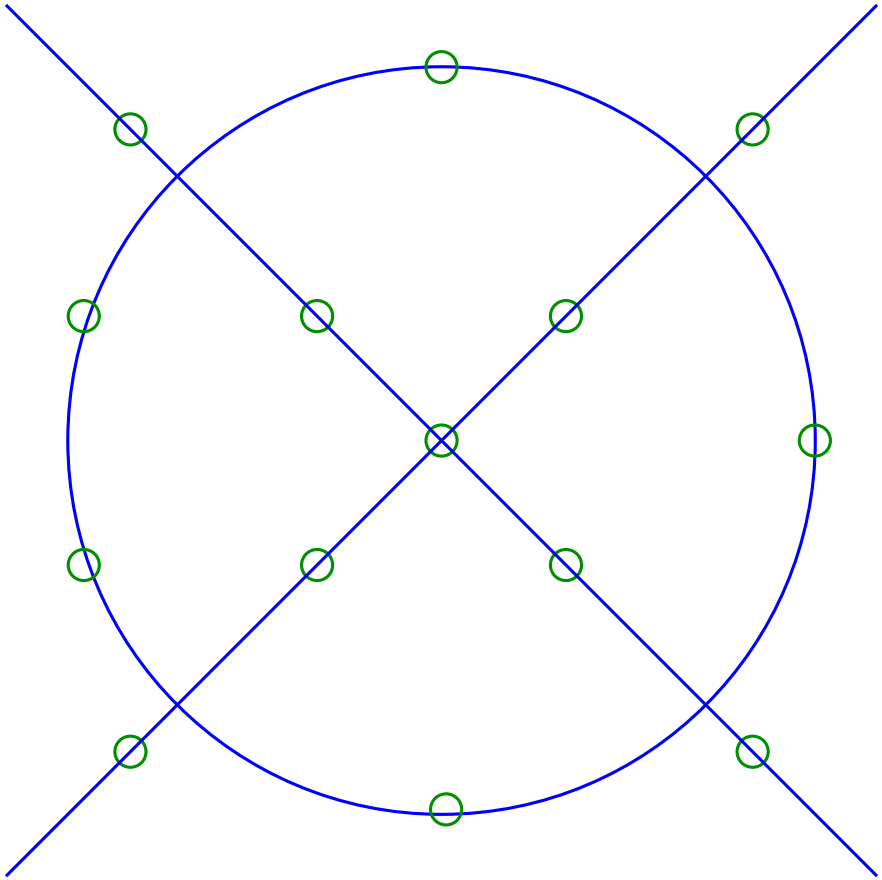
Find deg. 4 poly. $Q(x, y) \not\equiv 0$
s.t. $Q(\alpha_i, y_i) = 0$ for all points.

$$Q(x, y) = y^4 - x^4 - y^2 + x^2$$

Let us plot all zeroes of Q ...

Algorithm Description [S. '96]

$n = 14$ points; Want all *lines* through at least **5** points.



Find deg. 4 poly. $Q(x, y) \not\equiv 0$
s.t. $Q(\alpha_i, y_i) = 0$ for all points.

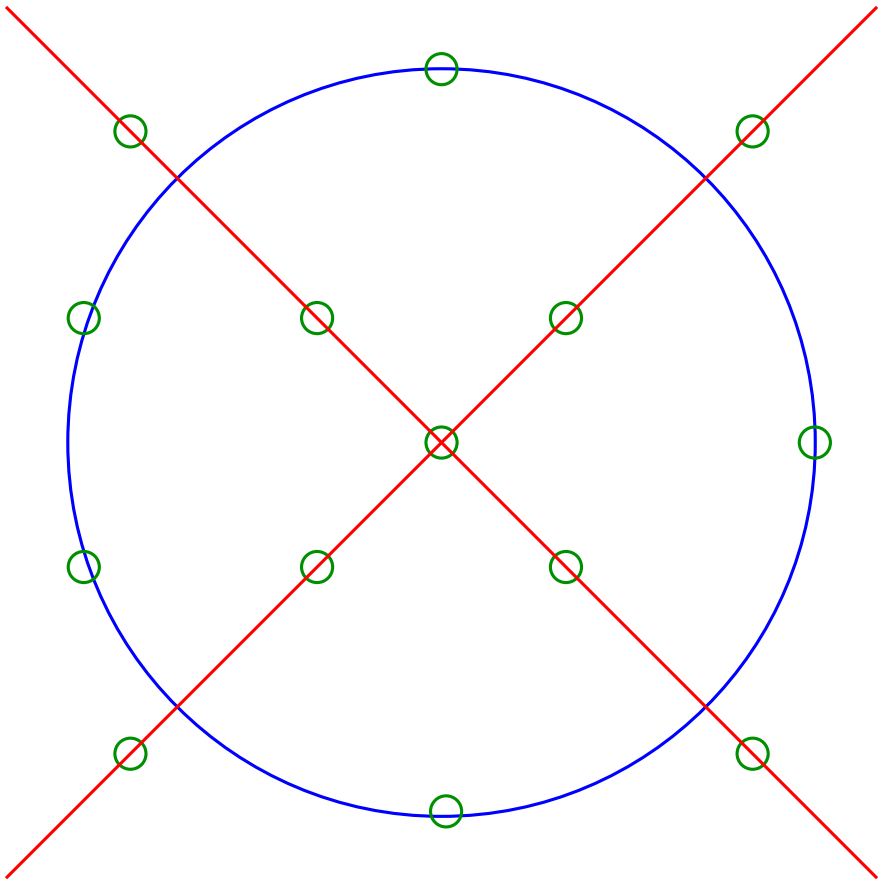
$$Q(x, y) = y^4 - x^4 - y^2 + x^2$$

Let us plot all zeroes of Q ...

Both relevant lines emerge !

Algorithm Description [S. '96]

$n = 14$ points; Want all *lines* through at least **5** points.



Find deg. 4 poly. $Q(x, y) \not\equiv 0$
s.t. $Q(\alpha_i, y_i) = 0$ for all points.

$$Q(x, y) = y^4 - x^4 - y^2 + x^2$$

Let us plot all zeroes of Q ...

Both relevant lines emerge !

Formally, $Q(x, y)$ factors as:
$(x^2 + y^2 - 1)(y + x)(y - x)$.

What Happened?

1. Why did degree 4 curve exist?
   - Counting argument: degree 4 gives enough degrees of freedom to pass through any 14 points.
2. Why did all the relevant lines emerge/factor out?
   - Line ℓ intersects a deg. 4 curve Q in 5 points $\Longrightarrow$ ℓ is a factor of Q

Generally

Lemma 1: $\exists Q$ with $\deg_x(Q), \deg_y(Q) \leq D = \sqrt{n}$ passing thru any n points.

Lemma 2: If Q with $\deg_x(Q), \deg_y(Q) \leq D$ intersects $y - p(x)$ with $\deg(p) \leq d$ intersect in more that $(D+1)d$ points, then $y - p(x)$ divides Q.

Efficient algorithm?

1. Can find Q by solving system of linear equations

Efficient algorithm?

1. Can find Q by solving system of linear equations
2. Fast algorithms for factorization of bivariate polynomials exist (’83-’85) [Kaltofen, Chistov & Grigoriev, Lenstra, von zur Gathen & Kaltofen]

Efficient algorithm?

1. Can find Q by solving system of linear equations
2. Fast algorithms for factorization of bivariate polynomials exist (’83-’85) [Kaltofen, Chistov & Grigoriev, Lenstra, von zur Gathen & Kaltofen]

- Immediate application:

Efficient algorithm?

1. Can find Q by solving system of linear equations
2. Fast algorithms for factorization of bivariate polynomials exist ('83-'85) [Kaltofen, Chistov & Grigoriev, Lenstra, von zur Gathen & Kaltofen]

- Immediate application:

  Theorem: Can list-decode Reed-Solomon code from $n - (k+1)\sqrt{n}$ errors.

Efficient algorithm?

1. Can find Q by solving system of linear equations
2. Fast algorithms for factorization of bivariate polynomials exist ('83-'85) [Kaltofen, Chistov & Grigoriev, Lenstra, von zur Gathen & Kaltofen]

- Immediate application:

  Theorem: Can list-decode Reed-Solomon code from $n - (k+1)\sqrt{n}$ errors.

- With some fine-tuning of parameters:

Efficient algorithm?

1. Can find Q by solving system of linear equations
2. Fast algorithms for factorization of bivariate polynomials exist ('83-'85) [Kaltofen, Chistov & Grigoriev, Lenstra, von zur Gathen & Kaltofen]

- Immediate application:

  Theorem: Can list-decode Reed-Solomon code from $n - (k+1)\sqrt{n}$ errors.

- With some fine-tuning of parameters:

  Theorem: [S. '96] Can list-decode Reed-Solomon code from $1 - \sqrt{2R}$-fraction errors.

Efficient algorithm?

1. Can find Q by solving system of linear equations
2. Fast algorithms for factorization of bivariate polynomials exist ('83-'85) [Kaltofen, Chistov & Grigoriev, Lenstra, von zur Gathen & Kaltofen]

- Immediate application:

  Theorem: Can list-decode Reed-Solomon code from $n - (k+1)\sqrt{n}$ errors.

- With some fine-tuning of parameters:

  Theorem: [S. '96] Can list-decode Reed-Solomon code from $1 - \sqrt{2R}$-fraction errors.

- Does not meet combinatorial bounds though!

Efficient algorithm?

1. Can find Q by solving system of linear equations
2. Fast algorithms for factorization of bivariate polynomials exist ('83-'85) [Kaltofen, Chistov & Grigoriev, Lenstra, von zur Gathen & Kaltofen]

- Immediate application:

  Theorem: Can list-decode Reed-Solomon code from $n - (k+1)\sqrt{n}$ errors.

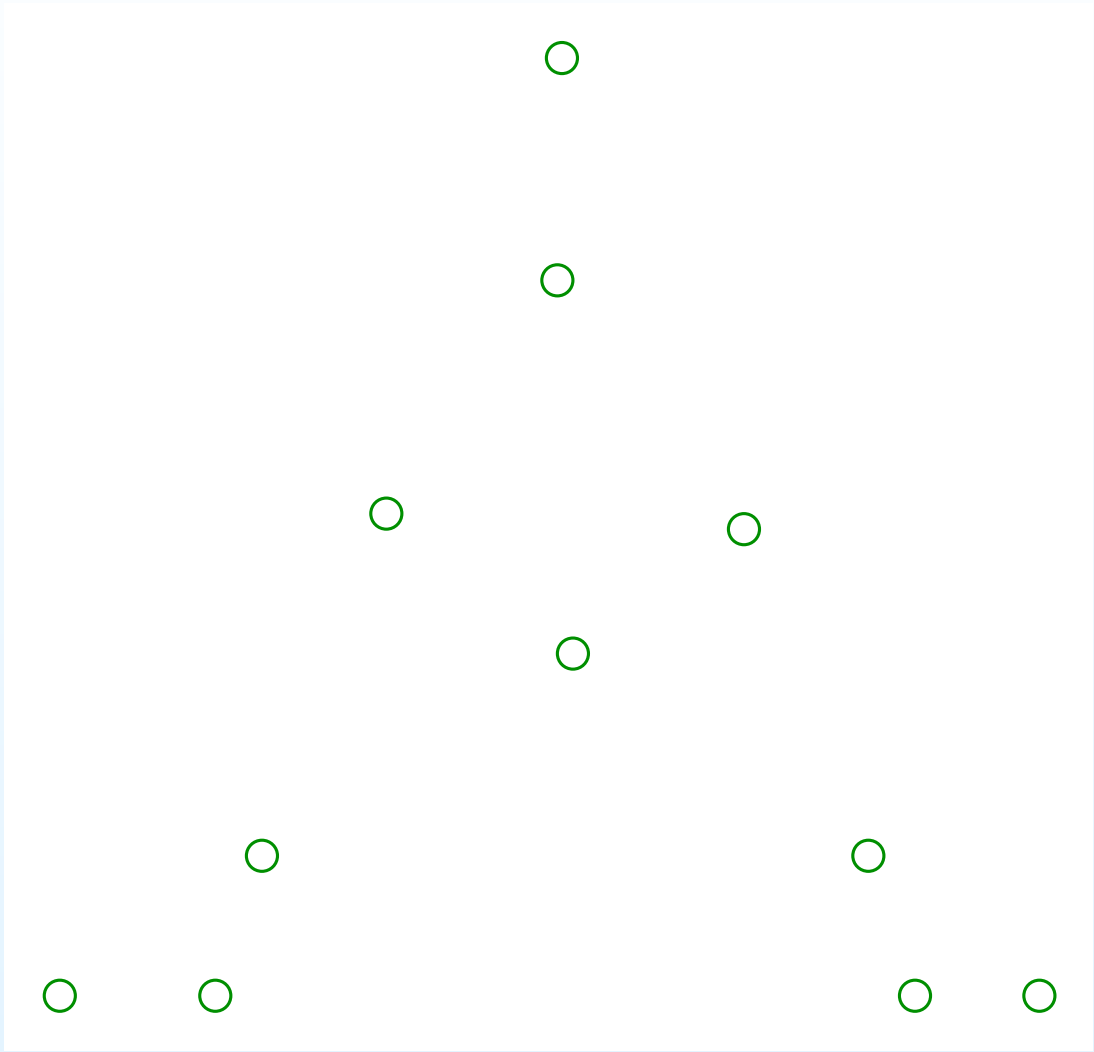
- With some fine-tuning of parameters:

  Theorem: [S. '96] Can list-decode Reed-Solomon code from $1 - \sqrt{2R}$-fraction errors.

- Does not meet combinatorial bounds though!

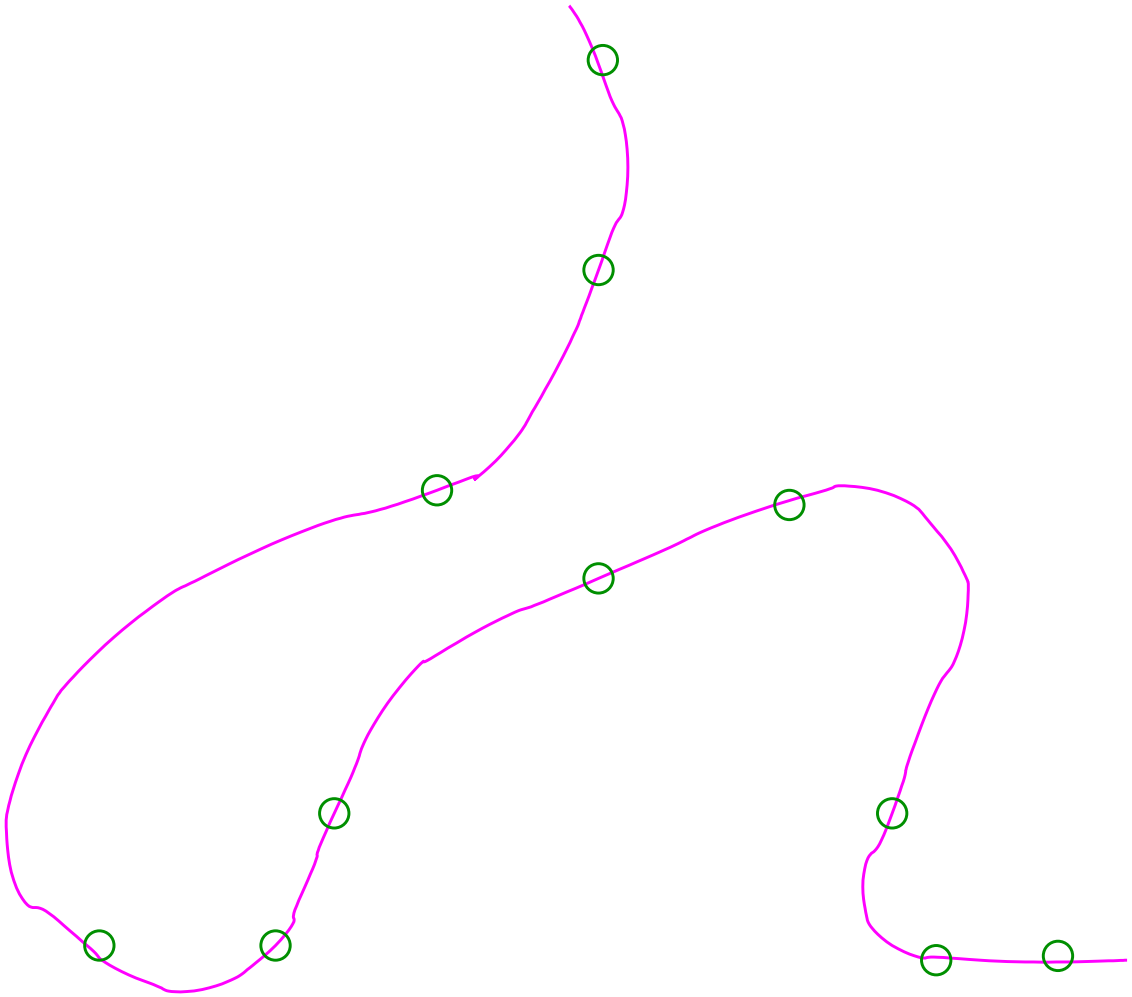
Improved List-Decoding

Going Further: Example 2 [Guruswami+S. '98]



$n = 11$ points; Want all lines through ≥ 4 pts.

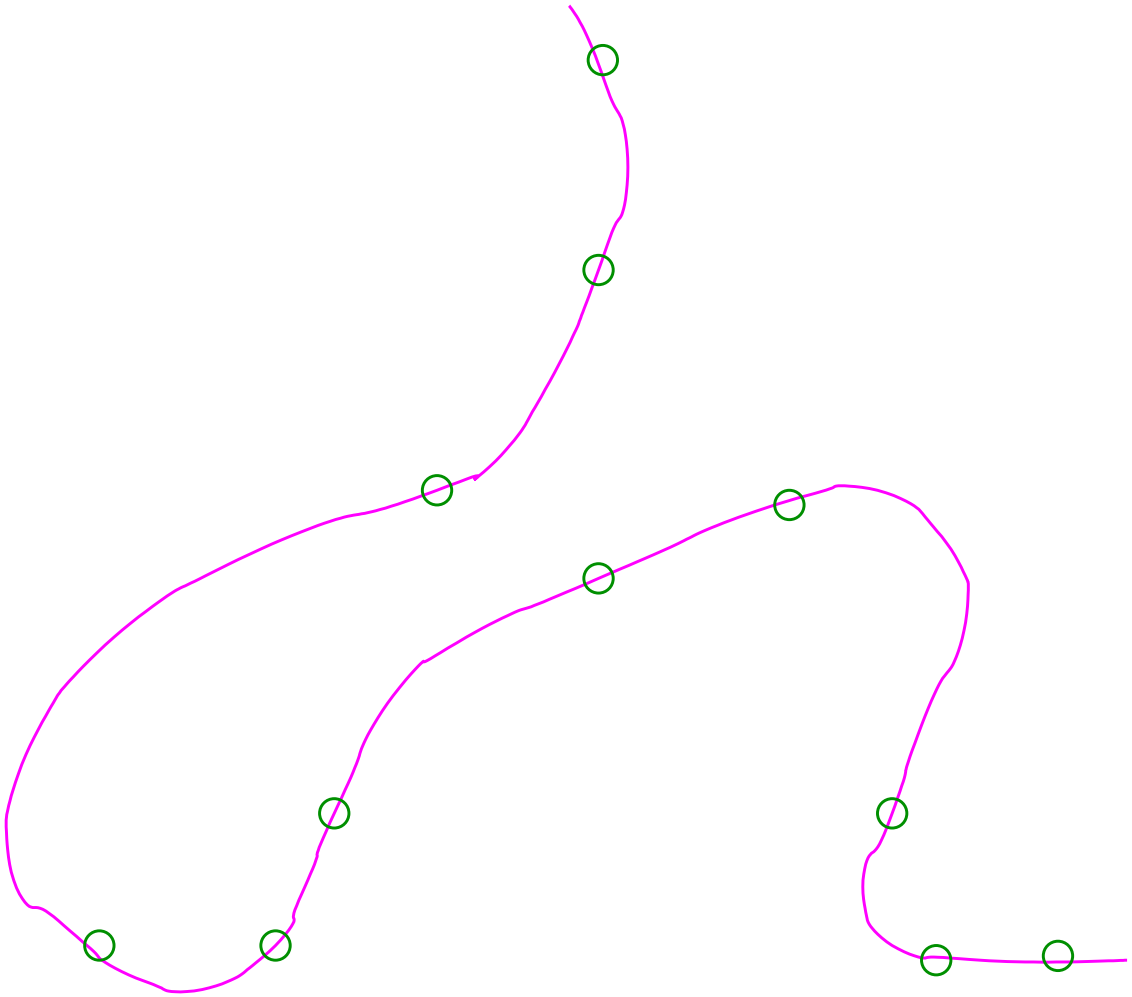
Going Further: Example 2 [Guruswami+S. '98]



$n = 11$ points; Want all lines through ≥ 4 pts.

Fitting degree 4 curve Q as earlier doesn't work.

Going Further: Example 2 [Guruswami+S. '98]

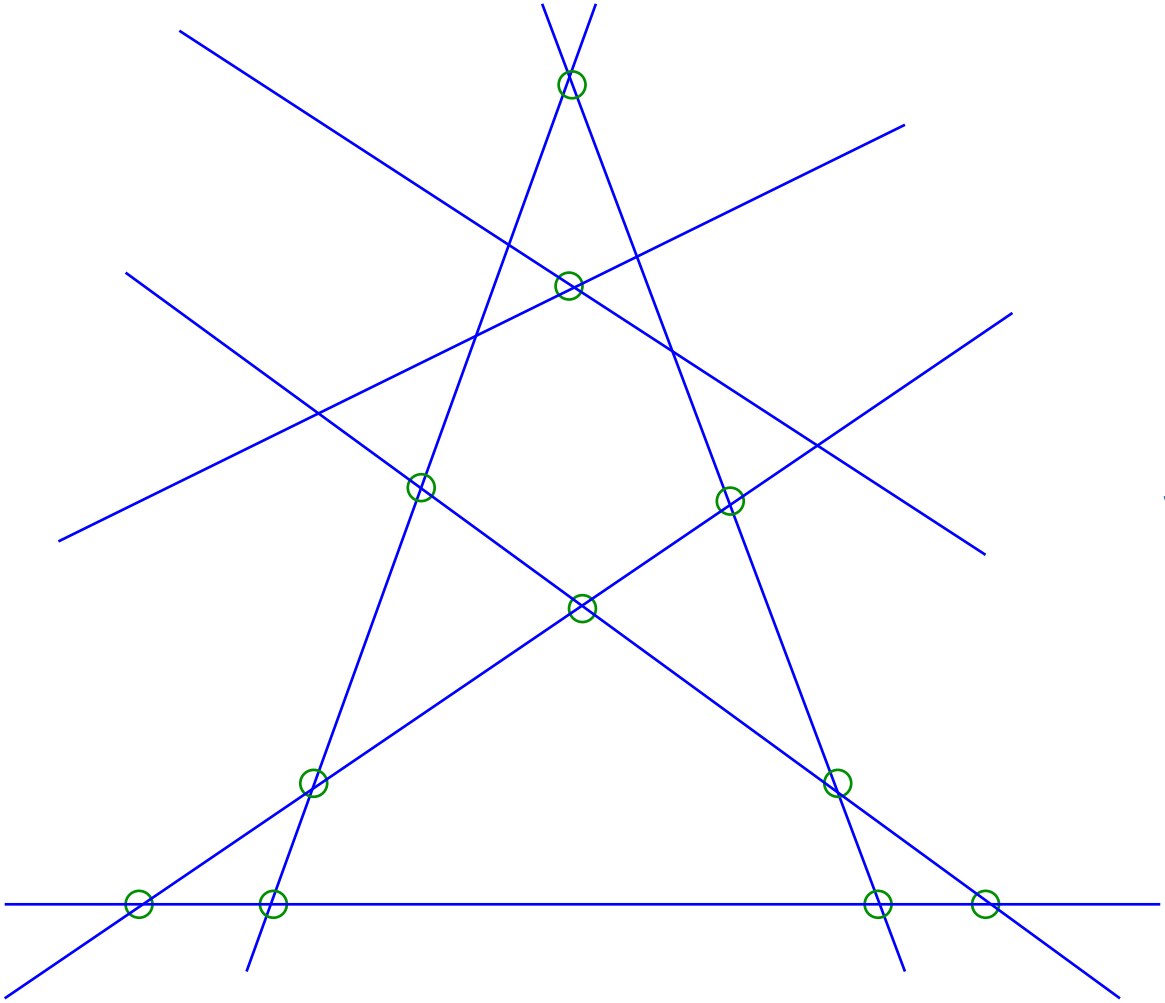


$n = 11$ points; Want all lines through ≥ 4 pts.

Fitting degree 4 curve Q as earlier doesn't work.

Why?

Going Further: Example 2 [Guruswami+S. '98]



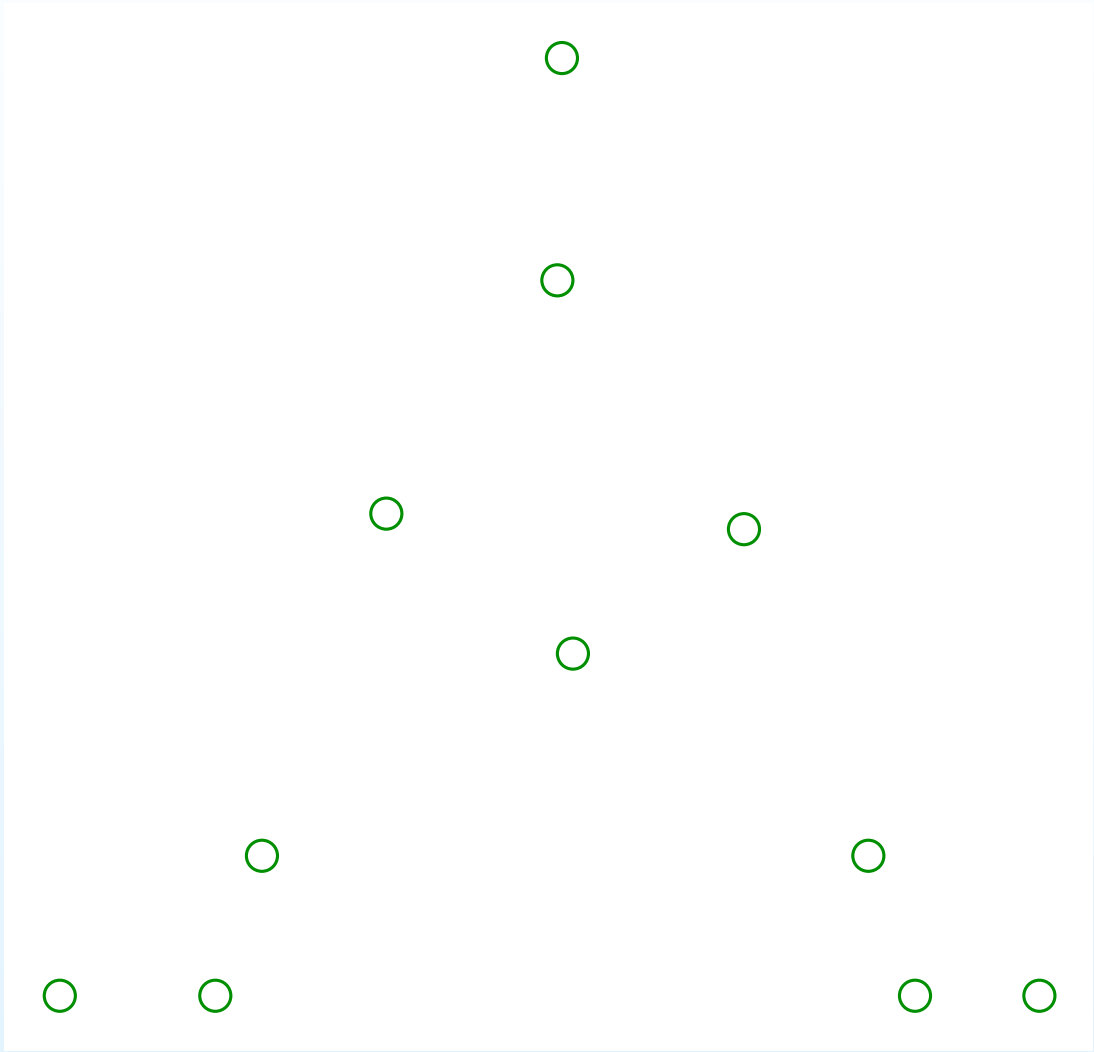
$n = 11$ points; Want all lines through ≥ 4 pts.

Fitting degree 4 curve Q as earlier doesn't work.

Why?

Correct answer has 5 lines.
Degree 4 curve can't have 5 factors!

Going Further: Example 2 [Guruswami+S. '98]



$n = 11$ points; Want all lines through ≥ 4 pts.

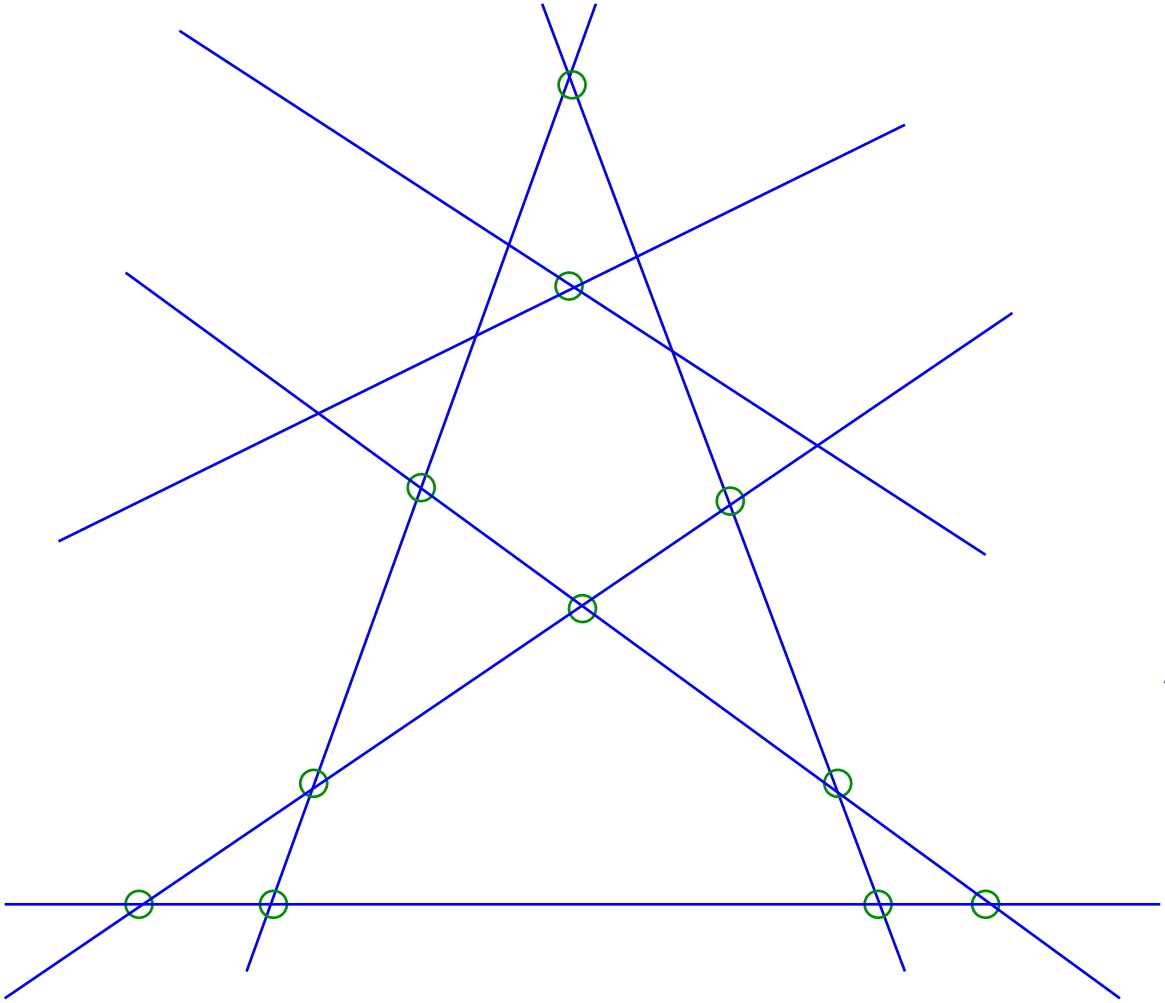
Fit degree 7 poly. $Q(x, y)$ passing through each point twice.

$Q(x, y) = \cdots$

(margin too small)

Plot all zeroes ...

Going Further: Example 2 [Guruswami+S. '98]



$n = 11$ points; Want all lines through ≥ 4 pts.

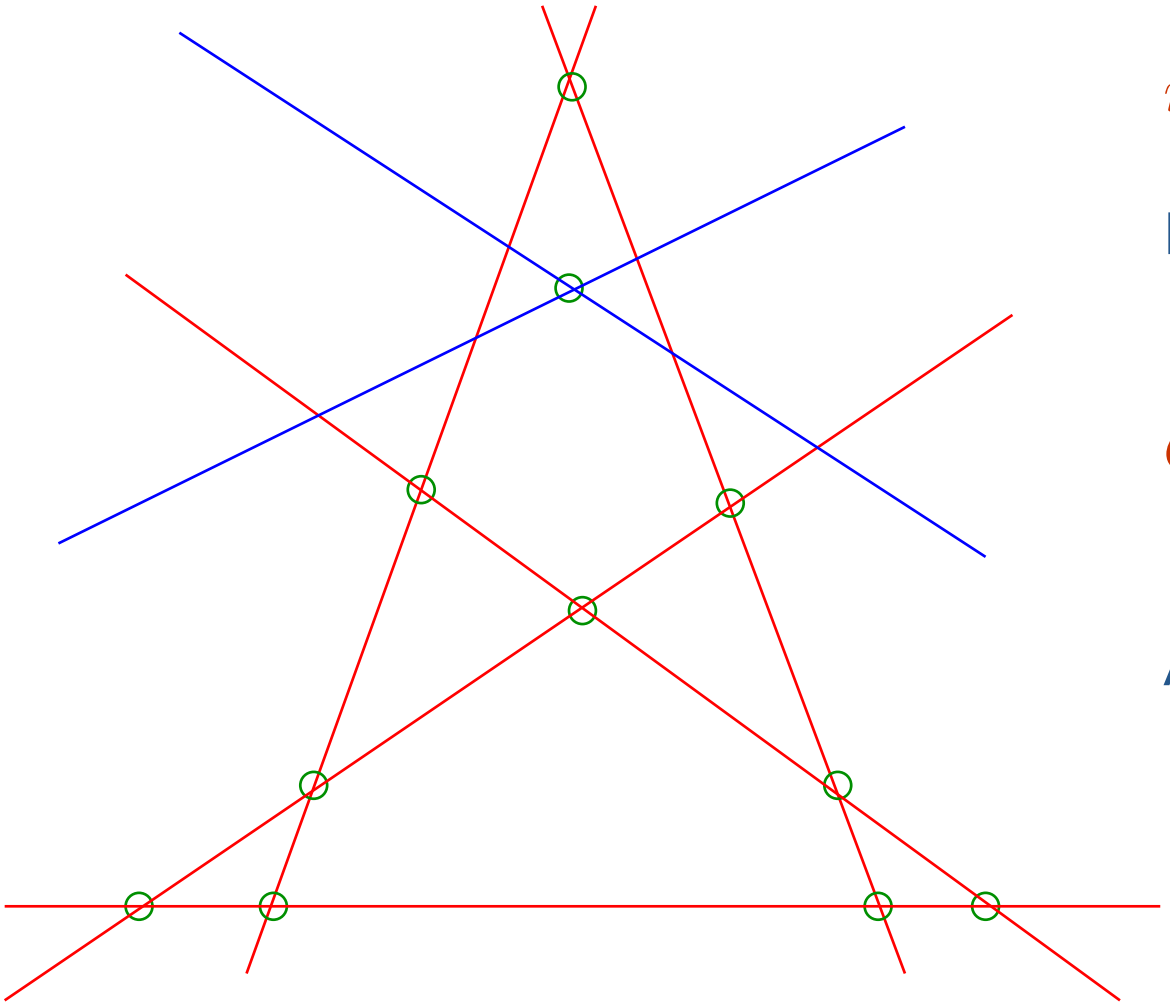
Fit degree 7 poly. $Q(x, y)$ passing through each point twice.

$Q(x, y) = \cdots$ (margin too small)

Plot all zeroes ...

All relevant lines emerge!

Going Further: Example 2 [Guruswami+S. '98]



$n = 11$ points; Want all lines through ≥ 4 pts.

Fit degree 7 poly. $Q(x, y)$ passing through each point twice.

$Q(x, y) = \cdots$ (margin too small)

Plot all zeroes ...

All relevant lines emerge!

Where was the gain?

- Requiring Q to pass through each point twice, effectively doubles the # intersections between Q and line.
  - So # intersections is now 8.
- On the other hand # constraints goes up from 11 to 33. Forces degree used to go upto 7 (from 4).
- But now # intersections is less than degree!

> Can pass through each point twice with less than twice the degree!

- Letting intersection multiplicity go to ∞ gives decoding algorithm for upto $1 - \sqrt{R}$ errors.

Summary

- Can correct errors in Reed-Solomon codes well beyond “half the distance” (Hamming) barrier!

Summary

- Can correct errors in Reed-Solomon codes well beyond “half the distance” (Hamming) barrier!
- Matches best known “combinatorial” bounds on list-decodability.

Summary

- Can correct errors in Reed-Solomon codes well beyond “half the distance” (Hamming) barrier!
- Matches best known “combinatorial” bounds on list-decodability.
- Open Question: Correct more errors, or show this leads to exponentially large lists!

Summary

- Can correct errors in Reed-Solomon codes well beyond “half the distance” (Hamming) barrier!
- Matches best known “combinatorial” bounds on list-decodability.
- Open Question: Correct more errors, or show this leads to exponentially large lists!
- Techniques: The polynomial method, and the method of multiplicities!

The Polynomial Method

- Goal: Understand some “combinatorial parameters” of some algebraically nice set. E.g.,

The Polynomial Method

- Goal: Understand some "combinatorial parameters" of some algebraically nice set. E.g.,
  - Minimum number of points in the union of ℓ sets where each set is t points from a degree k polynomial = ?
  - Minimum number of points in $K \subseteq \mathbb{F}_q^n$ such that K contains a line in every direction.

The Polynomial Method

- Goal: Understand some "combinatorial parameters" of some algebraically nice set. E.g.,
- Method:
  - Fit low-degree polynomial Q to the set K.
  - Infer Q is zero on points outside K, due to algebraic niceness.
  - Infer lower bound on degree of Q (due to abundance of zeroes).
  - Transfer to bound on combinatorial parameter of interest.

Kakeya Sets

- Definition: $K \subseteq \mathbb{F}_q^n$ is a Kakeya set if it contains a line in every direction.
- Question: How small can K be?

Kakeya Sets

- Definition: $K \subseteq \mathbb{F}_q^n$ is a Kakeya set if it contains a line in every direction.
- Question: How small can K be?
- Bounds (till 2007):
  $\forall K, \quad |K| \geq q^{n/2}$
  $\exists K, \quad |K| \leq q^n$

Kakeya Sets

- Definition: $K \subseteq \mathbb{F}_q^n$ is a Kakeya set if it contains a line in every direction.
- Question: How small can K be?
- Bounds (till 2007):
  $\forall K, \quad |K| \geq q^{n/2}$
  $\exists K, \quad |K| \leq\approx (q/2)^n$ [Mockenhaupt & Tao]

Kakeya Sets

- Definition: $K \subseteq \mathbb{F}_q^n$ is a Kakeya set if it contains a line in every direction.
- Question: How small can K be?
- Bounds (till 2007):
  $\forall K, \quad |K| \geq q^{n/2}$
  $\exists K, \quad |K| \leq\approx (q/2)^n$ [Mockenhaupt & Tao]
- In particular, even exponent of q unknown!

Kakeya Sets

- Definition: $K \subseteq \mathbb{F}_q^n$ is a Kakeya set if it contains a line in every direction.
- Question: How small can K be?
- Bounds (till 2007):
  $\forall K, \quad |K| \geq q^{n/2}$
  $\exists K, \quad |K| \leq\approx (q/2)^n$ [Mockenhaupt & Tao]
- In particular, even exponent of q unknown!
- [Dvir'08]'s breakthrough: $\forall K, \quad |K| \geq q^n/n!$

Kakeya Sets

- Definition: $K \subseteq \mathbb{F}_q^n$ is a Kakeya set if it contains a line in every direction.
- Question: How small can K be?
- Bounds (till 2007):
  $\forall K, \quad |K| \geq q^{n/2}$
  $\exists K, \quad |K| \leq\approx (q/2)^n$ [Mockenhaupt & Tao]
- In particular, even exponent of q unknown!
- [Dvir'08]'s breakthrough: $\forall K, \quad |K| \geq q^n/n!$
- Subsequently [Dvir, Kopparty, Saraf, S.]
  $\forall K, \quad |K| \geq (q/2)^n$

Polynomial Method and Kakeya Sets

- [Dvir'08]'s analysis:
  - Fit low-degree polynomial Q to K. (Interpolation $\Rightarrow$ Degree not too high if K not large.)

Polynomial Method and Kakeya Sets

- [Dvir'08]'s analysis:
  - Fit low-degree polynomial Q to K. (Interpolation $\Rightarrow$ Degree not too high if K not large.)
  - Show homogenous part of Q zero at y if line in direction y contained in K.

Polynomial Method and Kakeya Sets

- [Dvir'08]'s analysis:
  - Fit low-degree polynomial Q to K. (Interpolation $\Rightarrow$ Degree not too high if K not large.)
  - Show homogenous part of Q zero at y if line in direction y contained in K.
  - Conclude homogenous part is zero too often!

Polynomial Method and Kakeya Sets

- [Dvir'08]'s analysis:
  - Fit low-degree polynomial Q to K. (Interpolation $\Rightarrow$ Degree not too high if K not large.)
  - Show homogenous part of Q zero at y if line in direction y contained in K.
  - Conclude homogenous part is zero too often!
- [Saraf + S.], [Dvir + Kopparty + Saraf + S.]:
  - Fit Q to vanish many times at each point of K.
  - Yields better bounds!

Conclusions

- Importance of model of error.

Conclusions

- Importance of model of error.
- Virtues of relaxing some notions (e.g., list-decoding vs. unique-decoding)

Conclusions

- Importance of model of error.
- Virtues of relaxing some notions (e.g., list-decoding vs. unique-decoding)
- New algorithmic insights: Can be useful outside the context of list-decoding (e.g., [Koetter-Vardy] Soft-decision decoder).

Conclusions

- Importance of model of error.
- Virtues of relaxing some notions (e.g., list-decoding vs. unique-decoding)
- New algorithmic insights: Can be useful outside the context of list-decoding (e.g., [Koetter-Vardy] Soft-decision decoder).
- Central open question:

Conclusions

- Importance of model of error.
- Virtues of relaxing some notions (e.g., list-decoding vs. unique-decoding)
- New algorithmic insights: Can be useful outside the context of list-decoding (e.g., [Koetter-Vardy] Soft-decision decoder).
- Central open question:

  Constructive list-decodable *binary* codes of rate $1 - H(\rho)$ correcting ρ-fraction errors !!
  Corresponding question for large alphabets resolved by [ParvareshVardy05, GuruswamiRudra06].

Conclusions

- Importance of model of error.
- Virtues of relaxing some notions (e.g., list-decoding vs. unique-decoding)
- New algorithmic insights: Can be useful outside the context of list-decoding (e.g., [Koetter-Vardy] Soft-decision decoder).
- Central open question:

  Constructive list-decodable *binary* codes of rate $1 - H(\rho)$ correcting ρ-fraction errors !!
  Corresponding question for large alphabets resolved by [ParvareshVardy05, GuruswamiRudra06].
- New (?) mathematical insights.

Conclusions

- Importance of model of error.
- Virtues of relaxing some notions (e.g., list-decoding vs. unique-decoding)
- New algorithmic insights: Can be useful outside the context of list-decoding (e.g., [Koetter-Vardy] Soft-decision decoder).
- Central open question:

  Constructive list-decodable *binary* codes of rate $1 - H(\rho)$ correcting ρ-fraction errors !!
  Corresponding question for large alphabets resolved by [ParvareshVardy05, GuruswamiRudra06].
- New (?) mathematical insights.
- Challenge: Apply existing insights to other practical settings.

Thank You !!

Reed-Solomon Codes

by

Bernard Sklar

Introduction

In 1960, Irving Reed and Gus Solomon published a paper in the *Journal of the Society for Industrial and Applied Mathematics* [1]. This paper described a new class of error-correcting codes that are now called *Reed-Solomon* (R-S) *codes*. These codes have great power and utility, and are today found in many applications from compact disc players to deep-space applications. This article is an attempt to describe the paramount features of R-S codes and the fundamentals of how they work.

Reed-Solomon codes are *nonbinary cyclic* codes with symbols made up of m-bit sequences, where m is any positive integer having a value greater than 2. R-S (n, k) codes on m-bit symbols exist for all n and k for which

$$0 < k < n < 2^m + 2 \tag{1}$$

where k is the number of data symbols being encoded, and n is the total number of code symbols in the encoded block. For the most conventional R-S (n, k) code,

$$(n, k) = (2^m - 1, 2^m - 1 - 2t) \tag{2}$$

where t is the symbol-error correcting capability of the code, and $n - k = 2t$ is the number of parity symbols. An extended R-S code can be made up with $n = 2^m$ or $n = 2^m + 1$, but not any further.

Reed-Solomon codes achieve the *largest possible* code minimum distance for any linear code with the same encoder input and output block lengths. For nonbinary codes, the distance between two codewords is defined (analogous to Hamming distance) as the number of symbols in which the sequences differ. For Reed-Solomon codes, the code minimum distance is given by [2]

$$d_{\min} = n - k + 1 \tag{3}$$

The code is capable of correcting any combination of t or fewer errors, where t can be expressed as [3]

$$t = \left\lfloor \frac{d_{\min} - 1}{2} \right\rfloor = \left\lfloor \frac{n - k}{2} \right\rfloor \tag{4}$$

where $\lfloor x \rfloor$ means the largest integer not to exceed x. Equation (4) illustrates that for the case of R-S codes, correcting t symbol errors requires no more than $2t$ parity symbols. Equation (4) lends itself to the following intuitive reasoning. One can say that the decoder has $n - k$ redundant symbols to "spend," which is twice the amount of correctable errors. For each error, one redundant symbol is used to locate the error, and another redundant symbol is used to find its correct value.

The erasure-correcting capability, ρ, of the code is

$$\rho = d_{\min} - 1 = n - k \tag{5}$$

Simultaneous error-correction and erasure-correction capability can be expressed as follows:

$$2\alpha + \gamma < d_{\min} < n - k \tag{6}$$

where α is the number of symbol-error patterns that can be corrected and γ is the number of symbol erasure patterns that can be corrected. An advantage of nonbinary codes such as a Reed-Solomon code can be seen by the following comparison. Consider a binary $(n, k) = (7, 3)$ code. The entire n-tuple space contains $2^n = 2^7 = 128$ n-tuples, of which $2^k = 2^3 = 8$ (or 1/16 of the n-tuples) are codewords. Next, consider a *nonbinary* $(n, k) = (7, 3)$ code where each symbol is composed of $m = 3$ bits. The n-tuple space amounts to $2^{nm} = 2^{21} = 2{,}097{,}152$ n-tuples, of which $2^{km} = 2^9 = 512$ (or 1/4096 of the n-tuples) are codewords. When dealing with nonbinary symbols, each made up of m bits, only a small fraction (i.e., 2^{km} of the large number 2^{nm}) of possible n-tuples are codewords. This fraction decreases with increasing values of m. The important point here is that when a small fraction of the n-tuple space is used for codewords, a large $d_{\min}$ can be created.

Any linear code is capable of correcting $n - k$ symbol erasure patterns if the $n - k$ erased symbols all happen to lie on the parity symbols. However, R-S codes have the remarkable property that they are able to correct *any* set of $n - k$ symbol erasures within the block. R-S codes can be designed to have any redundancy. However, the complexity of a high-speed implementation increases with

redundancy. Thus, the most attractive R-S codes have high code rates (low redundancy).

Reed-Solomon Error Probability

The Reed-Solomon (R-S) codes are particularly useful for *burst-error correction*; that is, they are effective for channels that have memory. Also, they can be used efficiently on channels where the set of input symbols is large. An interesting feature of the R-S code is that as many as two information symbols can be added to an R-S code of length n without reducing its minimum distance. This extended R-S code has length $n + 2$ and the same number of parity check symbols as the original code. The R-S decoded symbol-error probability, P_E, in terms of the channel symbol-error probability, p, can be written as follows [4]:

$$P_E \approx \frac{1}{2^m - 1} \sum_{j=t+1}^{2^m-1} j \binom{2^m - 1}{j} p^j (1-p)^{2^m-1-j} \tag{7}$$

where t is the symbol-error correcting capability of the code, and the symbols are made up of m bits each.

The bit-error probability can be upper bounded by the symbol-error probability for specific modulation types. For MFSK modulation with $M = 2^m$, the relationship between P_B and P_E is as follows [3]:

$$\frac{P_B}{P_E} = \frac{2^{m-1}}{2^m - 1} \tag{8}$$

Figure 1 shows P_B versus the channel symbol-error probability p, plotted from Equations (7) and (8) for various (t-error-correcting 32-ary orthogonal Reed-Solomon codes with $n = 31$ (thirty-one 5-bit symbols per code block).

Figure 2 shows P_B versus E_b/N_0 for such a coded system using 32-ary MFSK modulation and noncoherent demodulation over an AWGN channel [4]. For R-S codes, error probability is an exponentially decreasing function of block length, n, and decoding complexity is proportional to a small power of the block length [2]. The R-S codes are sometimes used in a concatenated arrangement. In such a system, an inner convolutional decoder first provides some error control by operating on soft-decision demodulator outputs; the convolutional decoder then presents hard-decision data to the outer Reed-Solomon decoder, which further reduces the probability of error.

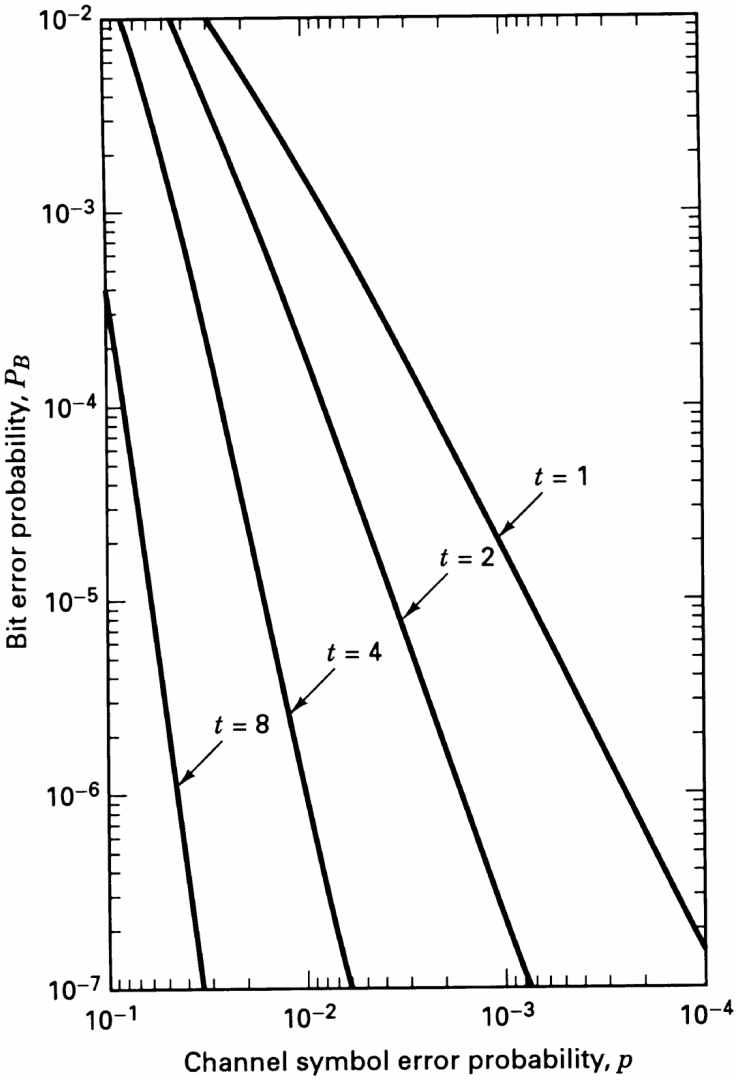


Figure 1

P_B versus p for 32-ary orthogonal signaling and $n = 31$, t-error correcting Reed-Solomon coding [4].

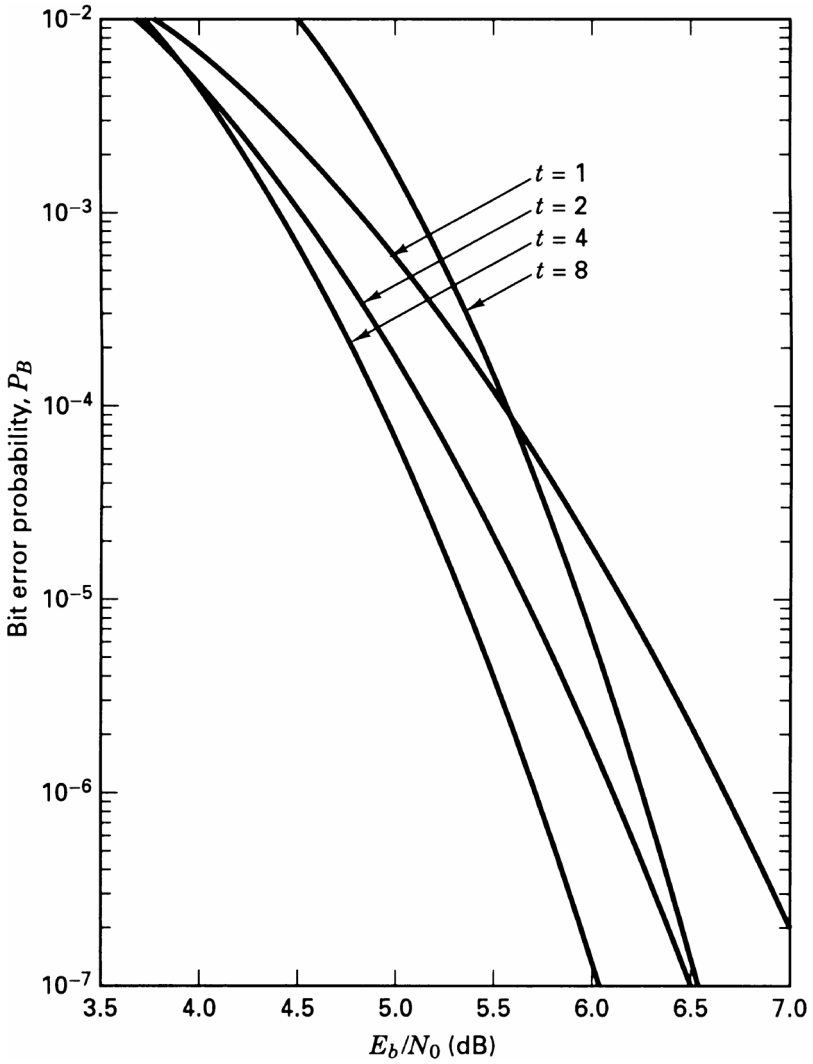


Figure 2

Bit-error probability versus E_b/N_0 performance of several $n = 31$, t-error correcting Reed-Solomon coding systems with 32-ary MPSK modulation over an AWGN channel [4].

Why R-S Codes Perform Well Against Burst Noise

Consider an (n, k) = (255, 247) R-S code, where each symbol is made up of m = 8 bits (such symbols are typically referred to as *bytes*). Since n - k = 8, Equation (4) indicates that this code can correct any four symbol errors in a block of 255. Imagine the presence of a noise burst, lasting for 25-bit durations and disturbing one block of data during transmission, as illustrated in Figure 3.

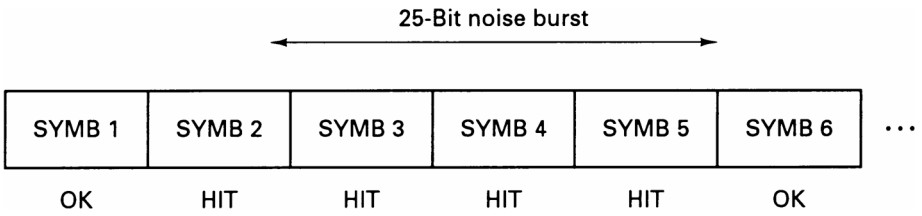


Figure 3

Data block disturbed by 25-bit noise burst.

In this example, notice that a burst of noise that lasts for a duration of 25 contiguous bits must disturb exactly four symbols. The R-S decoder for the (255, 247) code will correct *any* four-symbol errors without regard to the type of damage suffered by the symbol. In other words, when a decoder corrects a byte, it replaces the incorrect byte with the correct one, whether the error was caused by one bit being corrupted or all eight bits being corrupted. Thus if a symbol is wrong, it might as well be wrong in all of its bit positions. This gives an R-S code a tremendous burst-noise advantage over binary codes, even allowing for the interleaving of binary codes. In this example, if the 25-bit noise disturbance had occurred in a random fashion rather than as a contiguous burst, it should be clear that many more than four symbols would be affected (as many as 25 symbols might be disturbed). Of course, that would be beyond the capability of the (255, 247) code.

R-S Performance as a Function of Size, Redundancy, and Code Rate

For a code to successfully combat the effects of noise, the noise duration has to represent a relatively small percentage of the codeword. To ensure that this happens most of the time, the received noise should be averaged over a long period of time, reducing the effect of a freak streak of bad luck. Hence, error-correcting

codes become more efficient (error performance improves) as the code block size increases, making R-S codes an attractive choice whenever long block lengths are desired [5]. This is seen by the family of curves in Figure 4, where the rate of the code is held at a constant 7/8, while its block size increases from $n = 32$ symbols (with $m = 5$ bits per symbol) to $n = 256$ symbols (with $m = 8$ bits per symbol). Thus, the block size increases from 160 bits to 2048 bits.

As the redundancy of an R-S code increases (lower code rate), its implementation grows in complexity (especially for high-speed devices). Also, the bandwidth expansion must grow for any real-time communications application. However, the benefit of increased redundancy, just like the benefit of increased symbol size, is the improvement in bit-error performance, as can be seen in Figure 5, where the code length n is held at a constant 64, while the number of data symbols decreases from $k = 60$ to $k = 4$ (redundancy increases from 4 symbols to 60 symbols).

Figure 5 represents transfer functions (output bit-error probability versus input channel symbol-error probability) of hypothetical decoders. Because there is no system or channel in mind (only an output-versus-input of a decoder), you might get the idea that the improved error performance versus increased redundancy is a monotonic function that will continually provide system improvement even as the code rate approaches zero. However, this is not the case for codes operating in a real-time communication system. As the rate of a code varies from minimum to maximum (0 to 1), it is interesting to observe the effects shown in Figure 6. Here, the performance curves are plotted for BPSK modulation and an R-S (31, k) code for various channel types. Figure 6 reflects a real-time communication system, where the price paid for error-correction coding is bandwidth expansion by a factor equal to the inverse of the code rate. The curves plotted show clear optimum code rates that minimize the required E_b/N_0 [6]. The optimum code rate is about 0.6 to 0.7 for a Gaussian channel, 0.5 for a Rician-fading channel (with the ratio of direct to reflected received signal power, $K = 7$ dB), and 0.3 for a Rayleigh-fading channel. Why is there an E_b/N_0 degradation for very large rates (small redundancy) and very low rates (large redundancy)? It is easy to explain the degradation at high rates compared to the optimum rate. Any code generally provides a coding-gain benefit; thus, as the code rate approaches unity (no coding), the system will suffer worse error performance. The degradation at low code rates is more subtle because in a real-time communication system using both modulation and coding, there are two mechanisms at work. One mechanism works to improve error performance, and the other works to

degrade it. The improving mechanism is the coding; the greater the redundancy, the greater will be the error-correcting capability of the code. The degrading mechanism is the energy reduction per channel symbol (compared to the data symbol) that stems from the increased redundancy (and faster signaling in a real-time communication system). The reduced symbol energy causes the demodulator to make more errors. Eventually, the second mechanism wins out, and thus at very low code rates the system experiences error-performance degradation.

Let's see if we can corroborate the error performance versus code rate in Figure 6 with the curves in Figure 2. The figures are really not directly comparable because the modulation is BPSK in Figure 6 and 32-ary MFSK in Figure 2. However, perhaps we can verify that R-S error performance versus code rate exhibits the same general curvature with MFSK modulation as it does with BPSK. In Figure 2, the error performance over an AWGN channel improves as the symbol error-correcting capability, t, increases from $t = 1$ to $t = 4$; the $t = 1$ and $t = 4$ cases correspond to R-S (31, 29) and R-S (31, 23) with code rates of 0.94 and 0.74 respectively. However, at $t = 8$, which corresponds to R-S (31, 15) with code rate = 0.48, the error performance at $P_B = 10^{-5}$ degrades by about 0.5 dB of E_b/N_0 compared to the $t = 4$ case. From Figure 2, we can conclude that if we were to plot error performance versus code rate, the curve would have the same general "shape" as it does in Figure 6. Note that this manifestation cannot be gleaned from Figure 1, since that figure represents a decoder transfer function, which provides no information about the channel and the demodulation. Therefore, of the two mechanisms at work in the channel, the Figure 1 transfer function only presents the output-versus-input benefits of the decoder, and displays nothing about the loss of energy as a function of lower code rate.

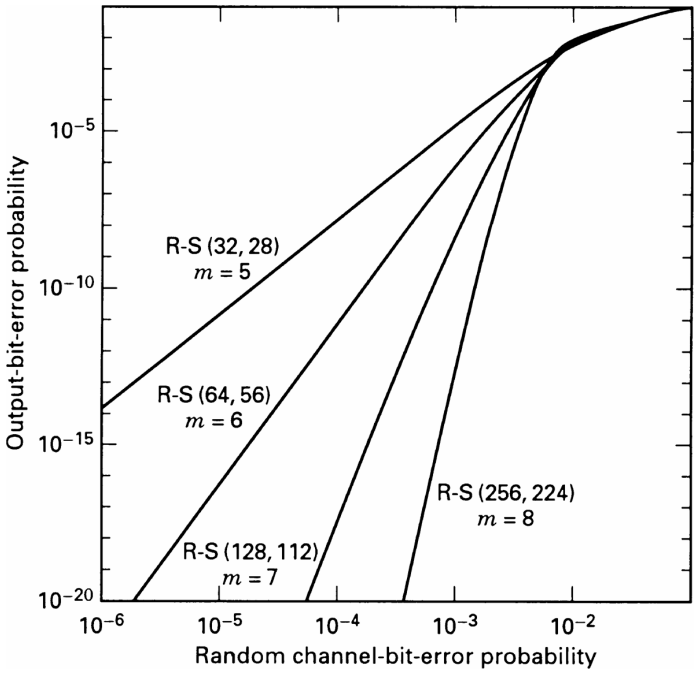


Figure 4

Reed-Solomon rate 7/8 decoder performance as a function of symbol size.

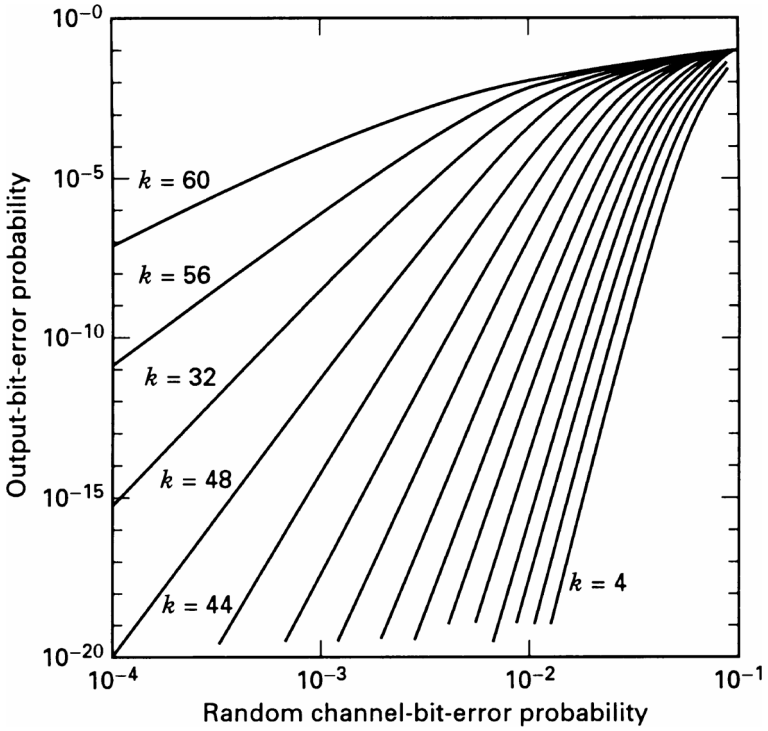


Figure 5

Reed-Solomon (64, *k*) decoder performance as a function of redundancy.

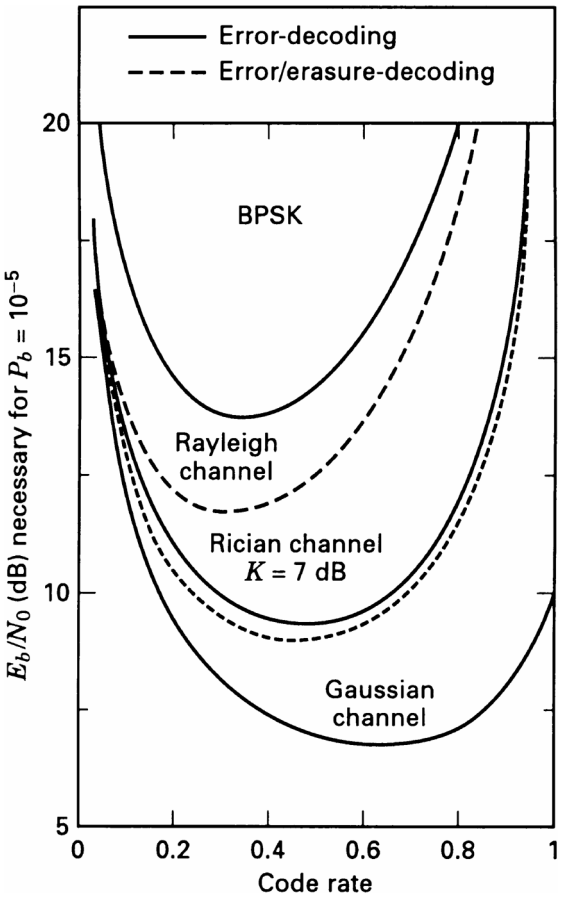


Figure 6

BPSK plus Reed-Solomon (31, *k*) decoder performance as a function of code rate.

Finite Fields

In order to understand the encoding and decoding principles of nonbinary codes, such as Reed-Solomon (R-S) codes, it is necessary to venture into the area of finite fields known as *Galois Fields* (GF). For any prime number, *p*, there exists a finite field denoted GF(*p*) that contains *p* elements. It is possible to extend GF(*p*) to a field of p^m elements, called an *extension field* of GF(*p*), and denoted by GF(p^m), where *m* is a nonzero positive integer. Note that GF(p^m) contains as a subset the elements of GF(*p*). Symbols from the extension field GF(2^m) are used in the construction of Reed-Solomon (R-S) codes.

The binary field GF(2) is a subfield of the extension field GF(2^m), in much the same way as the real number field is a subfield of the complex number field.

Besides the numbers 0 and 1, there are additional unique elements in the extension field that will be represented with a new symbol α. Each nonzero element in GF(2^m) can be represented by a power of α. An *infinite* set of elements, *F*, is formed by starting with the elements {0, 1, α}, and generating additional elements by progressively multiplying the last entry by α, which yields the following:

$$F = \{0, 1, \alpha, \alpha^2, \ldots, \alpha^j, \ldots\} = \{0, \alpha^0, \alpha^1, \alpha^2, \ldots, \alpha^j, \ldots\} \tag{9}$$

To obtain the *finite* set of elements of GF(2^m) from *F*, a condition must be imposed on *F* so that it may contain only 2^m elements and is closed under multiplication. The condition that closes the set of field elements under multiplication is characterized by the irreducible polynomial shown below:

$$\alpha^{(2^m-1)} + 1 = 0$$

or equivalently

$$\alpha^{(2^m-1)} = 1 = \alpha^0 \tag{10}$$

Using this polynomial constraint, any field element that has a power equal to or greater than 2^m - 1 can be reduced to an element with a power less than 2^m - 1, as follows:

$$\alpha^{(2^m+n)} = \alpha^{(2^m-1)}\ \alpha^{n+1} = \alpha^{n+1} \tag{11}$$

Thus, Equation (10) can be used to form the finite sequence *F** from the infinite sequence *F* as follows:

$$\begin{aligned} F^* &= \left\{ 0, 1, \alpha, \alpha^2, \ldots, \alpha^{2^m-2}, \alpha^{2^m-1}, \alpha^{2^m}, \ldots \right\} \\ &= \left\{ 0, \alpha^0, \alpha^1, \alpha^2, \ldots, \alpha^{2^m-2}, \alpha^0, \alpha^1, \alpha^2, \ldots \right\} \end{aligned} \tag{12}$$

Therefore, it can be seen from Equation (12) that the elements of the finite field, GF(2^m), are as follows:

$$\mathrm{GF}(2^m) = \left\{ 0, \alpha^0, \alpha^1, \alpha^2, \ldots, \alpha^{2^m-2} \right\} \tag{13}$$

Addition in the Extension Field GF(2^m)

Each of the 2^m elements of the finite field, GF(2^m), can be represented as a distinct polynomial of *degree m* - 1 or less. The degree of a polynomial is the value of its highest-order exponent. We denote each of the nonzero elements of GF(2^m) as a polynomial, $a_i(X)$, where at least one of the m coefficients of $a_i(X)$ is nonzero. For $i = 0,1,2,\ldots,2^m - 2$,

$$\alpha^i = a_i(X) = a_{i,0} + a_{i,1}X + a_{i,2}X^2 + \ldots + a_{i,m-1}X^{m-1} \tag{14}$$

Consider the case of $m = 3$, where the finite field is denoted GF(2^3). Figure 7 shows the mapping (developed later) of the seven elements $\{\alpha^i\}$ and the zero element, in terms of the basis elements $\{X^0, X^1, X^2\}$ described by Equation (14). Since Equation (10) indicates that $\alpha^0 = \alpha^7$, there are seven nonzero elements or a total of eight elements in this field. Each row in the Figure 7 mapping comprises a sequence of binary values representing the coefficients $a_{i,0}$, $a_{i,1}$, and $a_{i,2}$ in Equation (14). One of the benefits of using extension field elements $\{\alpha^i\}$ in place of binary elements is the compact notation that facilitates the mathematical representation of nonbinary encoding and decoding processes. Addition of two elements of the finite field is then defined as the modulo-2 sum of each of the polynomial coefficients of like powers,

$$\alpha^i + \alpha^j = (a_{i,0} + a_{j,0}) + (a_{i,1} + a_{j,1})X + \ldots + (a_{i,m-1} + a_{j,m-1})X^{m-1} \tag{15}$$

Field elements	Basis elements X^0	X^1	X^2
0	0	0	0
α^0	1	0	0
α^1	0	1	0
α^2	0	0	1
α^3	1	1	0
α^4	0	1	1
α^5	1	1	1
α^6	1	0	1
α^7	1	0	0

Figure 7

Mapping field elements in terms of basis elements for GF(8) with $f(x) = 1 + x + x^3$.

A Primitive Polynomial Is Used to Define the Finite Field

A class of polynomials called *primitive polynomials* is of interest because such functions define the finite fields GF(2^m) that in turn are needed to define R-S codes. The following condition is necessary and sufficient to guarantee that a polynomial is primitive. An irreducible polynomial $f(X)$ of degree m is said to be primitive if the smallest positive integer n for which $f(X)$ divides $X^n + 1$ is $n = 2^m - 1$. Note that the statement A divides B means that A divided into B yields a nonzero quotient and a zero remainder. Polynomials will usually be shown low order to high order. Sometimes, it is convenient to follow the reverse format (for example, when performing polynomial division).

Example 1: Recognizing a Primitive Polynomial

Based on the definition of a primitive polynomial given above, determine whether the following irreducible polynomials are primitive.

a. $1 + X + X^4$

b. $1 + X + X^2 + X^3 + X^4$

Solution

a. We can verify whether this degree $m = 4$ polynomial is primitive by determining whether it divides $X^n + 1 = X^{(2^m - 1)} + 1 = X^{15} + 1$, but does not divide $X^n + 1$, for values of n in the range of $1 \le n < 15$. It is easy to verify that $1 + X + X^4$ divides $X^{15} + 1$ [3], and after repeated computations it can be verified that $1 + X + X^4$ will not divide $X^n + 1$ for any n in the range of $1 \le n < 15$. Therefore, $1 + X + X^4$ is a primitive polynomial.

b. It is simple to verify that the polynomial $1 + X + X^2 + X^3 + X^4$ divides $X^{15} + 1$. Testing to see whether it will divide $X^n + 1$ for some n that is less than 15 yields the fact that it also divides $X^5 + 1$. Thus, although $1 + X + X^2 + X^3 + X^4$ is irreducible, it is not primitive.

The Extension Field GF(2^3)

Consider an example involving a primitive polynomial and the finite field that it defines. Table 1 contains a listing of some primitive polynomials. We choose the first one shown, $f(X) = 1 + X + X^3$, which defines a finite field GF(2^m), where the degree of the polynomial is $m = 3$. Thus, there are $2^m = 2^3 = 8$ elements in the field defined by $f(X)$. Solving for the roots of $f(X)$ means that the values of X that

correspond to $f(X) = 0$ must be found. The familiar binary elements, 1 and 0, do not satisfy (are not roots of) the polynomial $f(X) = 1 + X + X^3$, since $f(1) = 1$ and $f(0) = 1$ (using modulo-2 arithmetic). Yet, a fundamental theorem of algebra states that a polynomial of degree m must have precisely m roots. Therefore for this example, $f(X) = 0$ must yield three roots. Clearly a dilemma arises, since the three roots do not lie in the same finite field as the coefficients of $f(X)$. Therefore, they must lie somewhere else; the roots lie in the extension field, GF(2^3). Let α, an element of the extension field, be defined as a root of the polynomial $f(X)$. Therefore, it is possible to write the following:

$$f(\alpha) = 0$$

$$1 + \alpha + \alpha^3 = 0 \tag{16}$$

$$\alpha^3 = -1 - \alpha$$

Since in the binary field $+1 = -1$, α^3 can be represented as follows:

$$\alpha^3 = 1 + \alpha \tag{17}$$

Thus, α^3 is expressed as a weighted sum of α-terms having lower orders. In fact all powers of α can be so expressed. For example, consider α^4, where we obtain

$$\alpha^4 = \alpha \bullet \alpha^3 = \alpha \bullet (1 + \alpha) = \alpha + \alpha^2 \tag{18a}$$

Now, consider α^5, where

$$\alpha^5 = \alpha \bullet \alpha^4 = \alpha \bullet (\alpha + \alpha^2) = \alpha^2 + \alpha^3 \tag{18b}$$

From Equation (17), we obtain

$$\alpha^5 = 1 + \alpha + \alpha^2 \tag{18c}$$

Now, for α^6, using Equation (18c), we obtain

$$\alpha^6 = \alpha \bullet \alpha^5 = \alpha \bullet (1 + \alpha + \alpha^2) = \alpha + \alpha^2 + \alpha^3 = 1 + \alpha^2 \tag{18d}$$

And for α^7, using Equation (18d), we obtain

$$\alpha^7 = \alpha \bullet \alpha^6 = \alpha \bullet (1 + \alpha^2) = \alpha + \alpha^3 = 1 = \alpha^0 \tag{18e}$$

Note that $\alpha^7 = \alpha^0$, and therefore the eight finite field elements of GF(2^3) are

$$\{0, \alpha^0, \alpha^1, \alpha^2, \alpha^3, \alpha^4, \alpha^5, \alpha^6\} \tag{19}$$

Table 1

Some Primitive Polynomials

m	
3	$1 + X + X^3$
4	$1 + X + X^4$
5	$1 + X^2 + X^5$
6	$1 + X + X^6$
7	$1 + X^3 + X^7$
8	$1 + X^2 + X^3 + X^4 + X^8$
9	$1 + X^4 + X^9$
10	$1 + X^3 + X^{10}$
11	$1 + X^2 + X^{11}$
12	$1 + X + X^4 + X^6 + X^{12}$
13	$1 + X + X^3 + X^4 + X^{13}$

m	
14	$1 + X + X^6 + X^{10} + X^{14}$
15	$1 + X + X^{15}$
16	$1 + X + X^{3+} X^{12} + X^{16}$
17	$1 + X^3 + X^{17}$
18	$1 + X^7 + X^{18}$
19	$1 + X + X^{2+} X^5 + X^{19}$
20	$1 + X^3 + X^{20}$
21	$1 + X^2 + X^{21}$
22	$1 + X + X^{22}$
23	$1 + X^5 + X^{23}$
24	$1 + X + X^{2+} X^7 + X^{24}$

The mapping of field elements in terms of basis elements, described by Equation (14), can be demonstrated with the linear feedback shift register (LFSR) circuit shown in Figure 8. The circuit generates (with $m = 3$) the $2^m - 1$ nonzero elements of the field, and thus summarizes the findings of Figure 7 and Equations (17) through (19). Note that in Figure 8 the circuit feedback connections correspond to the coefficients of the polynomial $f(X) = 1 + X + X^3$, just like for binary cyclic codes [3]. By starting the circuit in any nonzero state, say 1 0 0, and performing a right-shift at each clock time, it is possible to verify that each of the field elements shown in Figure 7 (except the all-zeros element) will cyclically appear in the stages of the shift register. Two arithmetic operations, addition and multiplication, can be defined for this GF(2^3) finite field. Addition is shown in Table 2, and multiplication is shown in Table 3 for the nonzero elements only. The rules of addition follow from Equations (17) through (18e), and can be verified by noticing in Figure 7 that the sum of any field elements can be obtained by adding (modulo-2) the respective coefficients of their basis elements. The multiplication rules in Table 3 follow the usual procedure, in which the product of the field elements is obtained by adding their exponents modulo-($2^m - 1$), or for this case, modulo-7.

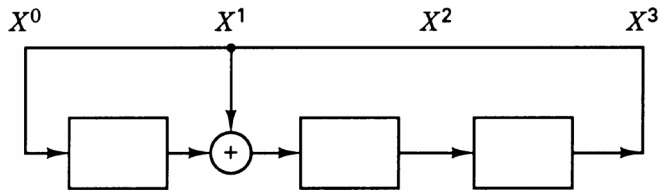


Figure 8

Extension field elements can be represented by the contents of a binary linear feedback shift register (LFSR) formed from a primitive polynomial.

Table 2

Addition Table

	α^0	α^1	α^2	α^3	α^4	α^5	α^6
α^0	0	α^3	α^6	α^1	α^5	α^4	α^2
α^1	α^3	0	α^4	α^0	α^2	α^6	α^5
α^2	α^6	α^4	0	α^5	α^1	α^3	α^0
α^3	α^1	α^0	α^5	0	α^6	α^2	α^4
α^4	α^5	α^2	α^1	α^6	0	α^0	α^3
α^5	α^4	α^6	α^3	α^2	α^0	0	α^1
α^6	α^2	α^5	α^0	α^4	α^3	α^1	0

Table 3

Multiplication Table

	α^0	α^1	α^2	α^3	α^4	α^5	α^6
α^0	α^0	α^1	α^2	α^3	α^4	α^5	α^6
α^1	α^1	α^2	α^3	α^4	α^5	α^6	α^0
α^2	α^2	α^3	α^4	α^5	α^6	α^0	α^1
α^3	α^3	α^4	α^5	α^6	α^0	α^1	α^2
α^4	α^4	α^5	α^6	α^0	α^1	α^2	α^3
α^5	α^5	α^6	α^0	α^1	α^2	α^3	α^4
α^6	α^6	α^0	α^1	α^2	α^3	α^4	α^5

A Simple Test to Determine Whether a Polynomial Is Primitive

There is another way of defining a primitive polynomial that makes its verification relatively easy. For an irreducible polynomial to be a primitive polynomial, at least one of its roots must be a primitive element. A *primitive element* is one that when raised to higher-order exponents will yield all the nonzero elements in the field. Since the field is a finite field, the number of such elements is finite.

Example 2: A Primitive Polynomial Must Have at Least One Primitive Element

Find the $m = 3$ roots of $f(X) = 1 + X + X^3$, and verify that the polynomial is primitive by checking that at least one of the roots is a primitive element. What are the roots? Which ones are primitive?

Solution

The roots will be found by enumeration. Clearly, $\alpha^0 = 1$ is not a root because $f(\alpha^0) = 1$. Now, use Table 2 to check whether α^1 is a root. Since

$$f(\alpha) = 1 + \alpha + \alpha^3 = 1 + \alpha^0 = 0$$

α is therefore a root.

Now check whether α^2 is a root:

$$f(\alpha^2) = 1 + \alpha^2 + \alpha^6 = 1 + \alpha^0 = 0$$

Hence, α^2 is a root.

Now check whether α^3 is a root.

$$f(\alpha^3) = 1 + \alpha^3 + \alpha^9 = 1 + \alpha^3 + \alpha^2 = 1 + \alpha^5 = \alpha^4 \neq 0$$

Hence, α^3 is *not* a root. Is α^4 a root?

$$f(\alpha^4) = \alpha^{12} + \alpha^4 + 1 = \alpha^5 + \alpha^4 + 1 = 1 + \alpha^0 = 0$$

Yes, it is a root. Hence, the roots of $f(X) = 1 + X + X^3$ are α, α^2, and α^4. It is not difficult to verify that starting with any of these roots and generating higher-order exponents yields all of the seven nonzero elements in the field. Hence, each of the roots is a primitive element. Since our verification requires that at least one root be a primitive element, the polynomial is primitive.

A relatively simple method to verify whether a polynomial is primitive can be described in a manner that is related to this example. For any given polynomial under test, draw the LFSR, with the feedback connections corresponding to the polynomial coefficients as shown by the example of Figure 8. Load into the circuit-registers any nonzero setting, and perform a right-shift with each clock pulse. If the circuit generates each of the nonzero field elements within one period, the polynomial that defines this GF(2^m) field is a primitive polynomial.

Reed-Solomon Encoding

Equation (2), repeated below as Equation (20), expresses the most conventional form of Reed-Solomon (R-S) codes in terms of the parameters n, k, t, and any positive integer $m > 2$.

$$(n, k) = (2^m - 1, 2^m - 1 - 2t) \tag{20}$$

where $n - k = 2t$ is the number of parity symbols, and t is the symbol-error correcting capability of the code. The generating polynomial for an R-S code takes the following form:

$$\mathbf{g}(X) = g_0 + g_1 X + g_2 X^2 + \ldots + g_{2t-1} X^{2t-1} + X^{2t} \tag{21}$$

The degree of the generator polynomial is equal to the number of parity symbols. R-S codes are a subset of the Bose, Chaudhuri, and Hocquenghem (BCH) codes; hence, it should be no surprise that this relationship between the degree of the generator polynomial and the number of parity symbols holds, just as for BCH codes. Since the generator polynomial is of degree $2t$, there must be precisely $2t$ successive powers of α that are roots of the polynomial. We designate the roots of $\mathbf{g}(X)$ as $\alpha, \alpha^2, \ldots, \alpha^{2t}$. It is not necessary to start with the root α; starting with any power of α is possible. Consider as an example the (7, 3) double-symbol-error correcting R-S code. We describe the generator polynomial in terms of its $2t = n - k = 4$ roots, as follows:

$$\begin{aligned}
\mathbf{g}(X) &= \left(X - \alpha\right)\left(X - \alpha^2\right)\left(X - \alpha^3\right)\left(X - \alpha^4\right) \\
&= \left(X^2 - \left(\alpha + \alpha^2\right)X + \alpha^3\right)\left(X^2 - \left(\alpha^3 + \alpha^4\right)X + \alpha^7\right) \\
&= \left(X^2 - \alpha^4 X + \alpha^3\right)\left(X^2 - \alpha^6 X + \alpha^0\right) \\
&= X^4 - \left(\alpha^4 + \alpha^6\right)X^3 + \left(\alpha^3 + \alpha^{10} + \alpha^0\right)X^2 - \left(\alpha^4 + \alpha^9\right)X + \alpha^3 \\
&= X^4 - \alpha^3 X^3 + \alpha^0 X^2 - \alpha^1 X + \alpha^3
\end{aligned}$$

Following the low order to high order format, and changing negative signs to positive, since in the binary field $+1 = -1$, $\mathbf{g}(X)$ can be expressed as follows:

$$\mathbf{g}(X) = \alpha^3 + \alpha^1 X + \alpha^0 X^2 + \alpha^3 X^3 + X^4 \tag{22}$$

Encoding in Systematic Form

Since R-S codes are cyclic codes, encoding in systematic form is analogous to the binary encoding procedure [3]. We can think of shifting a message polynomial, $\mathbf{m}(X)$, into the rightmost k stages of a codeword register and then appending a parity polynomial, $\mathbf{p}(X)$, by placing it in the leftmost $n - k$ stages. Therefore we multiply $\mathbf{m}(X)$ by X^{n-k}, thereby manipulating the message polynomial algebraically so that it is right-shifted $n - k$ positions. Next, we divide $X^{n-k}\,\mathbf{m}(X)$ by the generator polynomial $\mathbf{g}(X)$, which is written in the following form:

$$X^{n-k}\,\mathbf{m}(X) = \mathbf{q}(X)\,\mathbf{g}(X) + \mathbf{p}(X) \tag{23}$$

where $\mathbf{q}(X)$ and $\mathbf{p}(X)$ are quotient and remainder polynomials, respectively. As in the binary case, the remainder is the parity. Equation (23) can also be expressed as follows:

$$\mathbf{p}(X) = X^{n-k}\,\mathbf{m}(X) \text{ modulo } \mathbf{g}(X) \tag{24}$$

The resulting codeword polynomial, $\mathbf{U}(X)$ can be written as

$$\mathbf{U}(X) = \mathbf{p}(X) + X^{n-k}\,\mathbf{m}(X) \tag{25}$$

We demonstrate the steps implied by Equations (24) and (25) by encoding the following three-symbol message:

$$\underbrace{010}_{\alpha^1}\ \underbrace{110}_{\alpha^3}\ \underbrace{111}_{\alpha^5}$$

with the (7, 3) R-S code whose generator polynomial is given in Equation (22). We first multiply (upshift) the message polynomial $\alpha^1 + \alpha^3 X + \alpha^5 X^2$ by $X^{n-k} = X^4$, yielding $\alpha^1 X^4 + \alpha^3 X^5 + \alpha^5 X^6$. We next divide this upshifted message polynomial by the generator polynomial in Equation (22), $\alpha^3 + \alpha^1 X + \alpha^0 X^2 + \alpha^3 X^3 + X^4$. Polynomial division with nonbinary coefficients is more tedious than its binary counterpart, because the required operations of addition (subtraction) and multiplication (division) must follow the rules in Tables 2 and 3, respectively. It is left as an exercise for the reader to verify that this polynomial division results in the following remainder (parity) polynomial.

$$\mathbf{p}(X) = \alpha^0 + \alpha^2 X + \alpha^4 X^2 + \alpha^6 X^3$$

Then, from Equation (25), the codeword polynomial can be written as follows:

$$\mathbf{U}(X) = \alpha^0 + \alpha^2 X + \alpha^4 X^2 + \alpha^6 X^3 + \alpha^1 X^4 + \alpha^3 X^5 + \alpha^5 X^6$$

Systematic Encoding with an (n - k)–Stage Shift Register

Using circuitry to encode a three-symbol sequence in systematic form with the (7, 3) R-S code described by $\mathbf{g}(X)$ in Equation (22) requires the implementation of a linear feedback shift register (LFSR) circuit, as shown in Figure 9. It can easily be verified that the multiplier terms in Figure 9, taken from left to right, correspond to the coefficients of the polynomial in Equation (22) (low order to high order). This encoding process is the nonbinary equivalent of cyclic encoding [3]. Here, corresponding to Equation (20), the (7, 3) R-S nonzero codewords are made up of $2^m - 1 = 7$ symbols, and each symbol is made up of $m = 3$ bits.

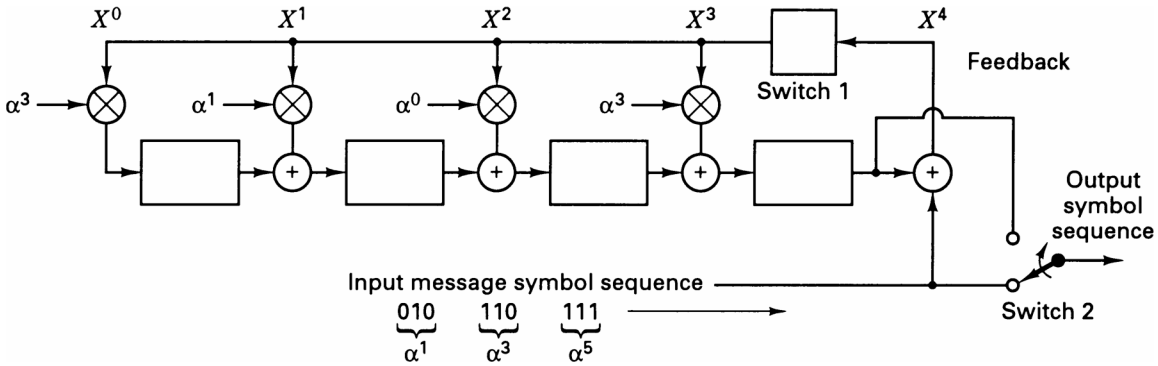


Figure 9

LFSR encoder for a (7, 3) R-S code.

Here the example is nonbinary, so that each stage in the shift register of Figure 9 holds a 3-bit symbol. In the case of binary codes, the coefficients labeled g_1, g_2, and so on are binary. Therefore, they take on values of 1 or 0, simply dictating the presence or absence of a connection in the LFSR. However in Figure 9, since each coefficient is specified by 3-bits, it can take on one of eight values.

The nonbinary operation implemented by the encoder of Figure 9, forming codewords in a systematic format, proceeds in the same way as the binary one. The steps can be described as follows:

1. Switch 1 is closed during the first k clock cycles to allow shifting the message symbols into the (n - k)–stage shift register.

2. Switch 2 is in the down position during the first k clock cycles in order to allow simultaneous transfer of the message symbols directly to an output register (not shown in Figure 9).

3. After transfer of the *k*th message symbol to the output register, switch 1 is opened and switch 2 is moved to the up position.

4. The remaining (n - k) clock cycles clear the parity symbols contained in the shift register by moving them to the output register.

5. The total number of clock cycles is equal to n, and the contents of the output register is the codeword polynomial $\mathbf{p}(X) + X^{n-k}\,\mathbf{m}(X)$, where $\mathbf{p}(X)$ represents the parity symbols and $\mathbf{m}(X)$ the message symbols in polynomial form.

We use the same symbol sequence that was chosen as a test message earlier:

$$\underbrace{010}_{\alpha^1}\ \underbrace{110}_{\alpha^3}\ \underbrace{111}_{\alpha^5}$$

where the rightmost symbol is the earliest symbol, and the rightmost bit is the earliest bit. The operational steps during the first $k = 3$ shifts of the encoding circuit of Figure 9 are as follows:

INPUT QUEUE			CLOCK CYCLE	REGISTER CONTENTS				FEEDBACK
α^1	α^3	α^5	0	0	0	0	0	α^5
	α^1	α^3	1	α^1	α^6	α^5	α^1	α^0
		α^1	2	α^3	0	α^2	α^2	α^4
		-	3	α^0	α^2	α^4	α^6	-

After the third clock cycle, the register contents are the four parity symbols, α^0, α^2, α^4, and α^6, as shown. Then, switch 1 of the circuit is opened, switch 2 is toggled to the up position, and the parity symbols contained in the register are shifted to the output. Therefore the output codeword, $\mathbf{U}(X)$, written in polynomial form, can be expressed as follows:

$$\mathbf{U}(X) = \sum_{n=0}^{6} u_n X^n$$

$$\begin{aligned}\mathbf{U}(X) &= \alpha^0 + \alpha^2 X + \alpha^4 X^2 + \alpha^6 X^3 + \alpha^1 X^4 + \alpha^3 X^5 + \alpha^5 X^6 \\ &= (100) + (001)\,X + (011)\,X^2 + (101)\,X^3 + (010)\,X^4 + (110)\,X^5 + (111)\,X^6\end{aligned} \tag{26}$$

The process of verifying the contents of the register at various clock cycles is somewhat more tedious than in the binary case. Here, the field elements must be added and multiplied by using Table 2 and Table 3, respectively.

The roots of a generator polynomial, $\mathbf{g}(X)$, must also be the roots of the codeword generated by $\mathbf{g}(X)$, because a valid codeword is of the following form:

$$\mathbf{U}(X) = \mathbf{m}(X)\,\mathbf{g}(X) \tag{27}$$

Therefore, an arbitrary codeword, when evaluated at any root of $\mathbf{g}(X)$, must yield zero. It is of interest to verify that the codeword polynomial in Equation (26) does indeed yield zero when evaluated at the four roots of $\mathbf{g}(X)$. In other words, this means checking that

$$\mathbf{U}(\alpha) = \mathbf{U}(\alpha^2) = \mathbf{U}(\alpha^3) = \mathbf{U}(\alpha^4) = \mathbf{0}$$

Evaluating each term independently yields the following:

$$\begin{aligned}
\mathbf{U}(\alpha) &= \alpha^0 + \alpha^3 + \alpha^6 + \alpha^9 + \alpha^5 + \alpha^8 + \alpha^{11} \\
&= \alpha^0 + \alpha^3 + \alpha^6 + \alpha^2 + \alpha^5 + \alpha^1 + \alpha^4 \\
&= \alpha^1 + \alpha^0 + \alpha^6 + \alpha^4 \\
&= \alpha^3 + \alpha^3 = \mathbf{0}
\end{aligned}$$

$$\begin{aligned}
\mathbf{U}(\alpha^2) &= \alpha^0 + \alpha^4 + \alpha^8 + \alpha^{12} + \alpha^9 + \alpha^{13} + \alpha^{17} \\
&= \alpha^0 + \alpha^4 + \alpha^1 + \alpha^5 + \alpha^2 + \alpha^6 + \alpha^3 \\
&= \alpha^5 + \alpha^6 + \alpha^0 + \alpha^3 \\
&= \alpha^1 + \alpha^1 = \mathbf{0}
\end{aligned}$$

$$\begin{aligned}
\mathbf{U}(\alpha^3) &= \alpha^0 + \alpha^5 + \alpha^{10} + \alpha^{15} + \alpha^{13} + \alpha^{18} + \alpha^{23} \\
&= \alpha^0 + \alpha^5 + \alpha^3 + \alpha^1 + \alpha^6 + \alpha^4 + \alpha^2 \\
&= \alpha^4 + \alpha^0 + \alpha^3 + \alpha^2 \\
&= \alpha^5 + \alpha^5 = \mathbf{0}
\end{aligned}$$

$$\begin{aligned}
\mathbf{U}(\alpha^4) &= \alpha^0 + \alpha^6 + \alpha^{12} + \alpha^{18} + \alpha^{17} + \alpha^{23} + \alpha^{29} \\
&= \alpha^0 + \alpha^6 + \alpha^5 + \alpha^4 + \alpha^3 + \alpha^2 + \alpha^1 \\
&= \alpha^2 + \alpha^0 + \alpha^5 + \alpha^1 \\
&= \alpha^6 + \alpha^6 = \mathbf{0}
\end{aligned}$$

This demonstrates the expected results that a codeword evaluated at any root of $\mathbf{g}(X)$ must yield zero.

Reed-Solomon Decoding

Earlier, a test message encoded in systematic form using a (7, 3) R-S code resulted in a codeword polynomial described by Equation (26). Now, assume that during transmission this codeword becomes corrupted so that two symbols are received in error. (This number of errors corresponds to the maximum error-correcting capability of the code.) For this seven-symbol codeword example, the error pattern, $\mathbf{e}(X)$, can be described in polynomial form as follows:

$$\mathbf{e}(X) = \sum_{n=0}^{6} e_n X^n \tag{28}$$

For this example, let the double-symbol error be such that

$$\begin{aligned} \mathbf{e}(X) &= 0 + 0X + 0X^2 + \alpha^2 X^3 + \alpha^5 X^4 + 0X^5 + 0X^6 \\ &= (000) + (000)X + (000)X^2 + (001)X^3 + (111)X^4 + (000)X^5 + (000)X^6 \end{aligned} \tag{29}$$

In other words, one parity symbol has been corrupted with a 1-bit error (seen as α^2), and one data symbol has been corrupted with a 3-bit error (seen as α^5). The received corrupted-codeword polynomial, $\mathbf{r}(X)$, is then represented by the sum of the transmitted-codeword polynomial and the error-pattern polynomial as follows:

$$\mathbf{r}(X) = \mathbf{U}(X) + \mathbf{e}(X) \tag{30}$$

Following Equation (30), we add $\mathbf{U}(X)$ from Equation (26) to $\mathbf{e}(X)$ from Equation (29) to yield $\mathbf{r}(X)$, as follows:

$$\begin{aligned} \mathbf{r}(X) &= (100) + (001)X + (011)X^2 + (100)X^3 + (101)X^4 + (110)X^5 + (111)X^6 \\ &= \alpha^0 + \alpha^2 X + \alpha^4 X^2 + \alpha^0 X^3 + \alpha^6 X^4 + \alpha^3 X^5 + \alpha^5 X^6 \end{aligned} \tag{31}$$

In this example, there are four unknowns—two error locations and two error values. Notice an important difference between the nonbinary decoding of $\mathbf{r}(X)$ that we are faced with in Equation (31) and binary decoding; in binary decoding, the decoder only needs to find the error locations [3]. Knowledge that there is an error at a particular location dictates that the bit must be "flipped" from 1 to 0 or vice versa. But here, the nonbinary symbols require that we not only learn the error locations, but also determine the correct symbol values at those locations. Since there are four unknowns in this example, four equations are required for their solution.

Syndrome Computation

The *syndrome* is the result of a parity check performed on **r** to determine whether **r** is a valid member of the codeword set [3]. If in fact **r** is a member, the syndrome **S** has value **0**. Any nonzero value of **S** indicates the presence of errors. Similar to the binary case, the syndrome **S** is made up of $n - k$ symbols, $\{S_i\}$ (i = 1, … , $n - k$). Thus, for this (7, 3) R-S code, there are four symbols in every syndrome vector; their values can be computed from the received polynomial, $\mathbf{r}(X)$. Note how the computation is facilitated by the structure of the code, given by Equation (27) and rewritten below:

$$\mathbf{U}(X) = \mathbf{m}(X)\,\mathbf{g}(X)$$

From this structure it can be seen that every valid codeword polynomial $\mathbf{U}(X)$ is a multiple of the generator polynomial $\mathbf{g}(X)$. Therefore, the roots of $\mathbf{g}(X)$ must also be the roots of $\mathbf{U}(X)$. Since $\mathbf{r}(X) = \mathbf{U}(X) + \mathbf{e}(X)$, then $\mathbf{r}(X)$ evaluated at each of the roots of $\mathbf{g}(X)$ should yield zero only when it is a valid codeword. Any errors will result in one or more of the computations yielding a nonzero result. The computation of a syndrome symbol can be described as follows:

$$S_i = \mathbf{r}(X)\Big|_{X=\alpha^i} = \mathbf{r}(\alpha^i) \qquad i = 1,\cdots,n-k \tag{32}$$

where $\mathbf{r}(X)$ contains the postulated two-symbol errors as shown in Equation (29). If $\mathbf{r}(X)$ were a valid codeword, it would cause each syndrome symbol S_i to equal 0. For this example, the four syndrome symbols are found as follows:

$$\begin{aligned} S_1 &= \mathbf{r}(\alpha) = \alpha^0 + \alpha^3 + \alpha^6 + \alpha^3 + \alpha^{10} + \alpha^8 + \alpha^{11} \\ &= \alpha^0 + \alpha^3 + \alpha^6 + \alpha^3 + \alpha^2 + \alpha^1 + \alpha^4 \\ &= \alpha^3 \end{aligned} \tag{33}$$

$$\begin{aligned} S_2 &= \mathbf{r}(\alpha^2) = \alpha^0 + \alpha^4 + \alpha^8 + \alpha^6 + \alpha^{14} + \alpha^{13} + \alpha^{17} \\ &= \alpha^0 + \alpha^4 + \alpha^1 + \alpha^6 + \alpha^0 + \alpha^6 + \alpha^3 \\ &= \alpha^5 \end{aligned} \tag{34}$$

$$\begin{aligned} S_3 &= \mathbf{r}(\alpha^3) = \alpha^0 + \alpha^5 + \alpha^{10} + \alpha^9 + \alpha^{18} + \alpha^{18} + \alpha^{23} \\ &= \alpha^0 + \alpha^5 + \alpha^3 + \alpha^2 + \alpha^4 + \alpha^4 + \alpha^2 \\ &= \alpha^6 \end{aligned} \tag{35}$$

$$\begin{aligned} S_4 &= \mathbf{r}(\alpha^4) = \alpha^0 + \alpha^6 + \alpha^{12} + \alpha^{12} + \alpha^{22} + \alpha^{23} + \alpha^{29} \\ &= \alpha^0 + \alpha^6 + \alpha^5 + \alpha^5 + \alpha^1 + \alpha^2 + \alpha^1 \\ &= 0 \end{aligned} \tag{36}$$

The results confirm that the received codeword contains an error (which we inserted), since $\mathbf{S} \neq \mathbf{0}$.

Example 3: A Secondary Check on the Syndrome Values

For the (7, 3) R-S code example under consideration, the error pattern is known, since it was chosen earlier. An important property of codes when describing the standard array is that each element of a coset (row) in the standard array has the same syndrome [3]. Show that this property is also true for the R-S code by evaluating the error polynomial $\mathbf{e}(X)$ at the roots of $\mathbf{g}(X)$ to demonstrate that it must yield the same syndrome values as when $\mathbf{r}(X)$ is evaluated at the roots of $\mathbf{g}(X)$. In other words, it must yield the same values obtained in Equations (33) through (36).

Solution

$$S_i = \mathbf{r}(X)\Big|_{X=\alpha^i} = \mathbf{r}(\alpha^i) \qquad i = 1, 2, \cdots, n-k$$

$$S_i = \left[\mathbf{U}(X) + \mathbf{e}(X) \right] \Big|_{X=\alpha^i} = \mathbf{U}(\alpha^i) + \mathbf{e}(\alpha^i)$$

$$S_i = \mathbf{r}(\alpha^i) = \mathbf{U}(\alpha^i) + \mathbf{e}(\alpha^i) = 0 + \mathbf{e}(\alpha^i)$$

From Equation (29),

$$\mathbf{e}(X) = \alpha^2 X^3 + \alpha^5 X^4$$

Therefore,

$$\begin{aligned} S_1 &= \mathbf{e}(\alpha^1) = \alpha^5 + \alpha^9 \\ &= \alpha^5 + \alpha^2 \\ &= \alpha^3 \\ S_2 &= \mathbf{e}(\alpha^2) = \alpha^8 + \alpha^{13} \\ &= \alpha^1 + \alpha^6 \\ &= \alpha^5 \end{aligned}$$

continues ➢

➢ *continued*

$$S_3 = \mathbf{e}(\alpha^3) = \alpha^{11} + \alpha^{17}$$
$$= \alpha^4 + \alpha^3$$
$$= \alpha^6$$
$$S_4 = \mathbf{e}(\alpha^4) = \alpha^{14} + \alpha^{21}$$
$$= \alpha^0 + \alpha^0$$
$$= 0$$

These results confirm that the syndrome values are the same, whether obtained by evaluating $\mathbf{e}(X)$ at the roots of $\mathbf{g}(X)$, or $\mathbf{r}(X)$ at the roots of $\mathbf{g}(X)$.

Error Location

Suppose there are ν errors in the codeword at location X^{j_1}, X^{j_2}, ... , X^{j_ν}. Then, the error polynomial $\mathbf{e}(X)$ shown in Equations (28) and (29) can be written as follows:

$$\mathbf{e}(X) = e_{j_1} X^{j_1} + e_{j_2} X^{j_2} + \ldots + e_{j_\nu} X^{j_\nu} \tag{37}$$

The indices 1, 2, … ν refer to the first, second, …, νth errors, and the index j refers to the error location. To correct the corrupted codeword, each error value e_{j_l} and its location X^{j_l}, where l = 1, 2, ..., ν, must be determined. We define an error locator number as $\beta_l = \alpha^{j_l}$. Next, we obtain the n - k = 2t syndrome symbols by substituting α^i into the received polynomial for i = 1, 2, … 2t:

$$S_1 = \mathbf{r}(\alpha) = e_{j_1}\beta_1 + e_{j_2}\beta_2 + \ldots + e_{j_\nu}\beta_\nu$$
$$S_2 = \mathbf{r}(\alpha^2) = e_{j_1}\beta_1^2 + e_{j_2}\beta_2^2 + \ldots + e_{j_\nu}\beta_\nu^2 \tag{38}$$
$$\vdots$$
$$S_{2t} = \mathbf{r}(\alpha^{2t}) = e_{j_1}\beta_1^{2t} + e_{j_2}\beta_2^{2t} + \ldots + e_{j_\nu}\beta_\nu^{2t}$$

There are $2t$ unknowns (t error values and t locations), and $2t$ simultaneous equations. However, these $2t$ simultaneous equations cannot be solved in the usual way because they are nonlinear (as some of the unknowns have exponents). Any technique that solves this system of equations is known as a *Reed-Solomon decoding algorithm*.

Once a nonzero syndrome vector (one or more of its symbols are nonzero) has been computed, that signifies that an error has been received. Next, it is necessary to learn the location of the error or errors. An error-locator polynomial, $\boldsymbol{\sigma}(X)$, can be defined as follows:

$$\begin{aligned}\boldsymbol{\sigma}(X) &= (\ 1\ +\ \beta_1 X)\ (\ 1\ +\ \beta_2 X)\ \ldots\ (\ 1\ +\ \beta_\nu X) \\ &= 1\ +\ \sigma_1 X +\ \sigma_2 X^2\ +\ldots+\sigma_\nu X^\nu\end{aligned} \tag{39}$$

The roots of $\boldsymbol{\sigma}(X)$ are $1/\beta_1$, $1/\beta_2$, … ,$1/\beta_\nu$. The reciprocal of the roots of $\boldsymbol{\sigma}(X)$ are the error-location numbers of the error pattern $\mathbf{e}(X)$. Then, using autoregressive modeling techniques [7], we form a matrix from the syndromes, where the first t syndromes are used to predict the next syndrome. That is,

$$\begin{bmatrix} S_1 & S_2 & S_3 & \cdots & S_{t-1} & S_t \\ S_2 & S_3 & S_4 & \cdots & S_t & S_{t+1} \\ & & & \bullet & & \\ & & & \bullet & & \\ & & & \bullet & & \\ S_{t-1} & S_t & S_{t+1} & \cdots & S_{2t-3} & S_{2t-2} \\ S_t & S_{t+1} & S_{t+2} & \cdots & S_{2t-2} & S_{2t-1} \end{bmatrix} \begin{bmatrix} \sigma_t \\ \sigma_{t-1} \\ \bullet \\ \bullet \\ \bullet \\ \sigma_2 \\ \sigma_1 \end{bmatrix} = \begin{bmatrix} -S_{t+1} \\ -S_{t+2} \\ \bullet \\ \bullet \\ \bullet \\ -S_{2t-1} \\ -S_{2t} \end{bmatrix} \tag{40}$$

We apply the autoregressive model of Equation (40) by using the largest dimensioned matrix that has a nonzero determinant. For the (7, 3) double-symbol-error correcting R-S code, the matrix size is 2 × 2, and the model is written as follows:

$$\begin{bmatrix} S_1 & S_2 \\ S_2 & S_3 \end{bmatrix} \begin{bmatrix} \sigma_2 \\ \sigma_1 \end{bmatrix} = \begin{bmatrix} S_3 \\ S_4 \end{bmatrix} \tag{41}$$

$$\begin{bmatrix} \alpha^3 & \alpha^5 \\ \alpha^5 & \alpha^6 \end{bmatrix} \begin{bmatrix} \sigma_2 \\ \sigma_1 \end{bmatrix} = \begin{bmatrix} \alpha^6 \\ 0 \end{bmatrix} \tag{42}$$

To solve for the coefficients $\boldsymbol{\sigma}_1$ and $\boldsymbol{\sigma}_2$ and of the error-locator polynomial, $\boldsymbol{\sigma}(X)$, we first take the inverse of the matrix in Equation (42). The inverse of a matrix $[A]$ is found as follows:

$$\text{Inv}\left[A\right] = \frac{\text{cofactor}\left[A\right]}{\det\left[A\right]}$$

Therefore,

$$\det \begin{bmatrix} \alpha^3 & \alpha^5 \\ \alpha^5 & \alpha^6 \end{bmatrix} = \alpha^3\alpha^6 - \alpha^5\alpha^5 = \alpha^9 + \alpha^{10} \tag{43}$$

$$= \alpha^2 + \alpha^3 = \alpha^5$$

$$\text{cofactor} \begin{bmatrix} \alpha^3 & \alpha^5 \\ \alpha^5 & \alpha^6 \end{bmatrix} = \begin{bmatrix} \alpha^6 & \alpha^5 \\ \alpha^5 & \alpha^3 \end{bmatrix} \tag{44}$$

$$\text{Inv}\begin{bmatrix}\alpha^3 & \alpha^5\\ \alpha^5 & \alpha^6\end{bmatrix} = \frac{\begin{bmatrix}\alpha^6 & \alpha^5\\ \alpha^5 & \alpha^3\end{bmatrix}}{\alpha^5} = \alpha^{-5}\begin{bmatrix}\alpha^6 & \alpha^5\\ \alpha^5 & \alpha^3\end{bmatrix}$$

$$= \alpha^2\begin{bmatrix}\alpha^6 & \alpha^5\\ \alpha^5 & \alpha^3\end{bmatrix} = \begin{bmatrix}\alpha^8 & \alpha^7\\ \alpha^7 & \alpha^5\end{bmatrix} = \begin{bmatrix}\alpha^1 & \alpha^0\\ \alpha^0 & \alpha^5\end{bmatrix} \tag{45}$$

Safety Check

If the inversion was performed correctly, the multiplication of the original matrix by the inverted matrix should yield an identity matrix.

$$\begin{bmatrix}\alpha^3 & \alpha^5\\ \alpha^5 & \alpha^6\end{bmatrix}\begin{bmatrix}\alpha^1 & \alpha^0\\ \alpha^0 & \alpha^5\end{bmatrix} = \begin{bmatrix}\alpha^4 + \alpha^5 & \alpha^3 + \alpha^{10}\\ \alpha^6 + \alpha^6 & \alpha^5 + \alpha^{11}\end{bmatrix} = \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} \tag{46}$$

Continuing from Equation (42), we begin our search for the error locations by solving for the coefficients of the error-locator polynomial, $\boldsymbol{\sigma}(X)$.

$$\begin{bmatrix}\sigma_2\\ \sigma_1\end{bmatrix} = \begin{bmatrix}\alpha^1 & \alpha^0\\ \alpha^0 & \alpha^5\end{bmatrix}\begin{bmatrix}\alpha^6\\ 0\end{bmatrix} = \begin{bmatrix}\alpha^7\\ \alpha^6\end{bmatrix} = \begin{bmatrix}\alpha^0\\ \alpha^6\end{bmatrix} \tag{47}$$

From Equations (39) and (47), we represent $\boldsymbol{\sigma}(X)$ as shown below.

$$\begin{aligned}\boldsymbol{\sigma}(X) &= \alpha^0 + \sigma_1 X + \sigma_2 X^2\\ &= \alpha^0 + \alpha^6 X + \alpha^0 X^2\end{aligned} \tag{48}$$

The roots of $\boldsymbol{\sigma}(X)$ are the reciprocals of the error locations. Once these roots are located, the error locations will be known. In general, the roots of $\boldsymbol{\sigma}(X)$ may be one or more of the elements of the field. We determine these roots by exhaustive

testing of the $\boldsymbol{\sigma}(X)$ polynomial with each of the field elements, as shown below. Any element X that yields $\boldsymbol{\sigma}(X) = \mathbf{0}$ is a root, and allows us to locate an error.

$$\boldsymbol{\sigma}(\alpha^0) = \alpha^0 + \alpha^6 + \alpha^0 = \alpha^6 \neq \mathbf{0}$$

$$\boldsymbol{\sigma}(\alpha^1) = \alpha^0 + \alpha^7 + \alpha^2 = \alpha^2 \neq \mathbf{0}$$

$$\boldsymbol{\sigma}(\alpha^2) = \alpha^0 + \alpha^8 + \alpha^4 = \alpha^6 \neq \mathbf{0}$$

$$\boldsymbol{\sigma}(\alpha^2) = \alpha^0 + \alpha^8 + \alpha^4 = \alpha^6 \neq \mathbf{0}$$

$$\boldsymbol{\sigma}(\alpha^3) = \alpha^0 + \alpha^9 + \alpha^6 = \mathbf{0} \Rightarrow \text{ERROR}$$

$$\boldsymbol{\sigma}(\alpha^4) = \alpha^0 + \alpha^{10} + \alpha^8 = \mathbf{0} \Rightarrow \text{ERROR}$$

$$\boldsymbol{\sigma}(\alpha^5) = \alpha^0 + \alpha^{11} + \alpha^{10} = \alpha^2 \neq \mathbf{0}$$

$$\boldsymbol{\sigma}(\alpha^6) = \alpha^0 + \alpha^{12} + \alpha^{12} = \alpha^0 \neq \mathbf{0}$$

As seen in Equation (39), the error locations are at the inverse of the roots of the polynomial. Therefore $\boldsymbol{\sigma}(\alpha^3) = \mathbf{0}$ indicates that one root exits at $1/\beta_l = \alpha^3$. Thus, $\beta_l = 1/\alpha^3 = \alpha^4$. Similarly, $\boldsymbol{\sigma}(\alpha^4) = \mathbf{0}$ indicates that another root exits at $1/\beta_{l'} = \alpha^4$. Thus, $\beta_{l'} = 1/\alpha^4 = \alpha^3$, where l and l' refer to the first, second, …, ν^{th} error. Therefore, in this example, there are two-symbol errors, so that the error polynomial is of the following form:

$$\mathbf{e}(X) = e_{j_1} X^{j_1} + e_{j_2} X^{j_2} \tag{49}$$

The two errors were found at locations α^3 and α^4. Note that the indexing of the error-location numbers is completely arbitrary. Thus, for this example, we can designate the $\beta_l = \alpha^{j_l}$ values as $\beta_1 = \alpha^{j_1} = \alpha^3$ and $\beta_2 = \alpha^{j_2} = \alpha^4$.

Error Values

An error had been denoted e_{j_1}, where the index j refers to the error location and the index l identifies the l^{th} error. Since each error value is coupled to a particular location, the notation can be simplified by denoting e_{j_1}, simply as e_1. Preparing to determine the error values e_1 and e_2 associated with locations $\beta_1 = \alpha^3$ and $\beta_2 = \alpha^4$,

any of the four syndrome equations can be used. From Equation (38), let's use S_1 and S_2.

$$S_1 = \mathbf{r}(\alpha) = e_1\beta_1 + e_2\beta_2 \tag{50}$$

$$S_2 = \mathbf{r}(\alpha^2) = e_1\beta_1^2 + e_2\beta_2^2$$

We can write these equations in matrix form as follows:

$$\begin{bmatrix} \beta_1 & \beta_2 \\ \beta_1^2 & \beta_2^2 \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \end{bmatrix} = \begin{bmatrix} S_1 \\ S_2 \end{bmatrix} \tag{51}$$

$$\begin{bmatrix} \alpha^3 & \alpha^4 \\ \alpha^6 & \alpha^8 \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \end{bmatrix} = \begin{bmatrix} \alpha^3 \\ \alpha^5 \end{bmatrix} \tag{52}$$

To solve for the error values e_1 and e_2, the matrix in Equation (52) is inverted in the usual way, yielding

$$\text{Inv} \begin{bmatrix} \alpha^3 & \alpha^4 \\ \alpha^6 & \alpha^1 \end{bmatrix} = \frac{\begin{bmatrix} \alpha^1 & \alpha^4 \\ \alpha^6 & \alpha^3 \end{bmatrix}}{\alpha^3\alpha^1 - \alpha^6\alpha^4}$$

$$= \frac{\begin{bmatrix} \alpha^1 & \alpha^4 \\ \alpha^6 & \alpha^3 \end{bmatrix}}{\alpha^4 + \alpha^3} = \alpha^{-6} \begin{bmatrix} \alpha^1 & \alpha^4 \\ \alpha^6 & \alpha^3 \end{bmatrix} = \alpha^1 \begin{bmatrix} \alpha^1 & \alpha^4 \\ \alpha^6 & \alpha^3 \end{bmatrix} \tag{53}$$

$$= \begin{bmatrix} \alpha^2 & \alpha^5 \\ \alpha^7 & \alpha^4 \end{bmatrix} = \begin{bmatrix} \alpha^2 & \alpha^5 \\ \alpha^0 & \alpha^4 \end{bmatrix}$$

Now, we solve Equation (52) for the error values, as follows:

$$\begin{bmatrix} e_1 \\ e_2 \end{bmatrix} = \begin{bmatrix} \alpha^2 & \alpha^5 \\ \alpha^0 & \alpha^4 \end{bmatrix} \begin{bmatrix} \alpha^3 \\ \alpha^5 \end{bmatrix} = \begin{bmatrix} \alpha^5 + \alpha^{10} \\ \alpha^3 + \alpha^9 \end{bmatrix} = \begin{bmatrix} \alpha^5 + \alpha^3 \\ \alpha^3 + \alpha^2 \end{bmatrix} = \begin{bmatrix} \alpha^2 \\ \alpha^5 \end{bmatrix} \tag{54}$$

Correcting the Received Polynomial with Estimates of the Error Polynomial

From Equations (49) and (54), the estimated error polynomial is formed, to yield the following:

$$\begin{aligned}\hat{\mathbf{e}}(X) &= e_1 X^{j_1} + e_2 X^{j_2} \\ &= \alpha^2 X^3 + \alpha^5 X^4\end{aligned} \tag{55}$$

The demonstrated algorithm repairs the received polynomial, yielding an estimate of the transmitted codeword, and ultimately delivers a decoded message. That is,

$$\hat{\mathbf{U}}(X) = \mathbf{r}(X) + \hat{\mathbf{e}}(X) = \mathbf{U}(X) + \mathbf{e}(X) + \hat{\mathbf{e}}(X) \tag{56}$$

$$\mathbf{r}(X) = (100) + (001)X + (011)X^2 + (100)X^3 + (101)X^4 + (110)X^5 + (111)X^6$$

$$\hat{\mathbf{e}}(X) = (000) + (000)X + (000)X^2 + (001)X^3 + (111)X^4 + (000)X^5 + (000)X^6$$

$$\begin{aligned}\hat{\mathbf{U}}(X) &= (100) + (001)X + (011)X^2 + (101)X^3 + (010)X^4 + (110)X^5 + (111)X^6 \\ &= \alpha^0 + \alpha^2 X + \alpha^4 X^2 + \alpha^6 X^3 + \alpha^1 X^4 + \alpha^3 X^5 + \alpha^5 X^6\end{aligned} \tag{57}$$

Since the message symbols constitute the rightmost $k = 3$ symbols, the decoded message is

$$\underbrace{010}_{\alpha^1}\ \underbrace{110}_{\alpha^3}\ \underbrace{111}_{\alpha^5}$$

which is exactly the test message that was chosen earlier for this example. For further reading on R-S coding, see the collection of papers in reference [8].

Conclusion

In this article, we examined Reed-Solomon (R-S) codes, a powerful class of nonbinary block codes, particularly useful for correcting burst errors. Because coding efficiency increases with code length, R-S codes have a special attraction. They can be configured with long block lengths (in bits) with less decoding time

than other codes of similar lengths. This is because the decoder logic works with symbol-based rather than bit-based arithmetic. Hence, for 8-bit symbols, the arithmetic operations would all be at the byte level. This increases the complexity of the logic, compared with binary codes of the same length, but it also increases the throughput.

References

[1] Reed, I. S. and Solomon, G., "Polynomial Codes Over Certain Finite Fields," *SIAM Journal of Applied Math.*, vol. 8, 1960, pp. 300-304.

[2] Gallager, R. G., *Information Theory and Reliable Communication* (New York: John Wiley and Sons, 1968).

[3] Sklar, B., *Digital Communications: Fundamentals and Applications, Second Edition* (Upper Saddle River, NJ: Prentice-Hall, 2001).

[4] Odenwalder, J. P., *Error Control Coding Handbook*, Linkabit Corporation, San Diego, CA, July 15, 1976.

[5] Berlekamp, E. R., Peile, R. E., and Pope, S. P., "The Application of Error Control to Communications," *IEEE Communications Magazine*, vol. 25, no. 4, April 1987, pp. 44-57.

[6] Hagenauer, J., and Lutz, E., "Forward Error Correction Coding for Fading Compensation in Mobile Satellite Channels," *IEEE JSAC*, vol. SAC-5, no. 2, February 1987, pp. 215-225.

[7] Blahut, R. E., *Theory and Practice of Error Control Codes* (Reading, MA: Addison-Wesley, 1983).

[8] Wicker, S. B. and Bhargava, V. K., ed., *Reed-Solomon Codes and Their Applications* (Piscataway, NJ: IEEE Press, 1983).

About the Author

Bernard Sklar is the author of *Digital Communications: Fundamentals and Applications, Second Edition* (Prentice-Hall, 2001, ISBN 0-13-084788-7).

Reed-Solomon Codes

Reed-Solomon codes

In these notes we examine Reed-Solomon codes, from a computer science point of view.

Reed-Solomon codes can be used as both error-correcting and erasure codes. In the error-correcting setting, we wish to transmit a sequence of numbers over a noisy communication channel. The channel noise might cause the data sent to arrive corrupted. In the erasure setting, the channel might fail to send our message. For both cases, we handle the problem of noise by sending additional information beyond the original message. The data sent is an *encoding* of the original message. If the noise is small enough, the additional information will allow the original message to be recovered, through a *decoding* process.

Encoding

Let us suppose that we wish to transmit a sequence of numbers $b_0, b_1, \ldots, b_{d-1}$. To simplify things, we will assume that these numbers are in $GF(p)$, i.e., our arithmetic is all done modulo p. In practice, we want to reduce everything to bits, bytes, and words, so we will later discuss how to compute over fields more conducive to this setting, namely fields of the form $GF(2^r)$.

Our encoding will be a longer sequence of numbers $e_0, e_1, \ldots, e_{n-1}$, where we require that $p > n$. We derive the e sequence from our original b sequence by using the b sequence to define a polynomial P, which we evaluate at n points. There are several ways to do this; here are two straightforward ones:

- Let $P(x) = b_0 + b_1 x + b_2 x^2 + \ldots b_{d-1} x^{d-1}$. This representation is convenient since it requires no computation to define the polynomial. Our encoding would consist of the values $P(0), P(1), \ldots, P(n-1)$. (Actually, our encoding could be the evalution of P at any set of n points; we choose this set for convenience. Notice that we need $p > n$, or we can't choose n distinct points!)
- Let $P(x) = c_0 + c_1 x + c_2 x^2 + \ldots c_{d-1} x^{d-1}$ be such that $P(0) = b_0$, $P(1) = b_1$, $\ldots$, $P(d-1) = b_{d-1}$. Our encoding $e_0, e_1, \ldots, e_{n-1}$ would consist of the values $P(0), P(1), \ldots, P(n-1)$. Although this representation requires computing the appropriate polynomial P, it has the advantage that our original message is actually sent as part of the encoding. A code with this property is called a *systematic* code. Systematic codes can be useful; for example, if there happen to be no erasures or errors, we might immediately be able to get our message on the other end.

  The polynomial $P(x)$ can be found by using a technique known as Lagrange interpolation. We know that there is a unique polynomial of degree $d-1$ that passes through d points. (For example, two points uniquely determine a line.) Given d points $(a_0, b_0), \ldots, (a_{d-1}, b_{d-1})$, it is easy to check that

$$P(x) = \sum_{j=0}^{d-1} b_j \prod_{k \neq j} \frac{x - a_k}{a_j - a_k}$$

  is a polynomial of degree $d-1$ that passes through the points. (To check this, note that $\prod_{k \neq j} \frac{x-a_k}{a_j-a_k}$ is 1 when $x = a_j$ and 0 when $x = a_k$ for $k \neq j$.)

Note that in either case, the encoded message is just a set of values obtained from a polynomial. The important point is not which actual polynomial we use (as long as the sender and receiver agree!), but just that we use a polynomial that is uniquely determined by the data values.

We will also have to assume that the sender and receiver agree on a system so that when the encoded information e_i arrives at the receiver, the receiver knows it corresponds to the value $P(i)$. If all the information is sent and arrives in a fixed order, this of course is not a problem. In the case of erasures, we must assume that when an encoded value is missing, we know that it is missing. For example, when information is sent over a network, usually the value i is derived from a packet header; hence we know when we receive a value e_i which number i it corresponds to.

Decoding

Let us now consider what must be done at the end of the receiver. The receiver must determine the polynomial from the received values; once the polynomial is determined, the receiver can determine the original message values. (In the first approach above, the coefficients of the polynomial are the message values; in the second, given the polynomial the message is determined by computing $P(0)$, $P(1)$, etc.)

Let us first consider the easier case, where there are erasures, but no errors. Suppose that just d (correct) values $e_{j_1}, e_{j_2}, \ldots, e_{j_d}$ arrive at the receiver. No matter which d values, the receiver can determine the polynomial, just by using Lagrange interpolation! Note that the polynomial the receiver computes must match P, since there is a unique polynomial of degree $d-1$ passing through d points.

What if there are errors, instead of erasures? (By the way, if there are erasures and errors, notice that we can pretend an erasure is an error, just by filling the erased value with a random value!) This is much harder. When there were only erasures, the receiver knows which values they received and that they are all correct. Here the receiver has obtained n values, $f_0, f_1, \ldots, f_{n-1}$, but has no idea which ones are correct.

We show, however, that as long as at most k received values are in error, that is $f_j \neq e_j$ at most k times, then the original message can be determined whenever $k \leq \frac{n-d}{2}$. Notice that we can make n as large as we like. If we know a bound on error rate in advance, we can choose n accordingly. By making n sufficiently large, we can deal with error rates of up to 50%. (Of course, recall that we need $p > n$.)

The decoding algorithm we cover is due to Berlekamp and Welch. An important concept for the decoding is an *error polynomial*. An error polynomial $E(x)$ satifies $E(i) = 0$ if $f_i \neq e_i$. That is, the polynomial E marks the positions where we have received erroneous values. Without loss of generality, we can assume that E has degree k and is *monic* (that is, the leasing coefficient is 1), since we can choose E to be $\prod_{i:f_i \neq e_i}(x-i)$ if there are k errors, and throw extra terms $x-i$ in the product where f_i equals e_i if there are fewer than k errors.

Now consider the following interesting (and somewhat magical) setup: we claim that there exist polynomials $V(x)$ of degree at most $d-1+k$ and monic $W(x)$ of degree at most k satisfying

$$V(i) = f_i W(i).$$

Namely, we can let $W(x) = E(x)$, and $V(x) = P(x) \cdot E(x)$. It is easy to check that $V(i)$ and $W(i)$ are both 0 when $f_i \neq e_i$, and are both $e_i W(i)$ otherwise. So, if someone handed us these polynomials V and W, we could find $P(x)$, since $P(x) = V(x)/W(x)$.

There are two things left to check. First, we need to show how to find polynomials V and W satisfying $V(i) = f_i W(i)$. Second, we need to check that when we find these polynomials, we don't somehow find a wrong pair of polynomials that do not satisfy $V(x)/W(x) = P(x)$. For example, a priori we could find a polynomial D that was different from E!

First, we show that we can find an V and W efficiently. Let $v_0, v_1, \ldots, v_{d+k-1}$ be the coefficients of V and $w_0, w_1, \ldots, w_k$ be the coefficients of W. Note we can assume $w_k = 1$. Then the equations $V(i) = f_i W(i)$ give n linear equations in the coefficients of V and W, so that we have n equations and $d+2k \leq n$ unknowns. Hence a pair V and W can be found by solving a set of linear equations.

Since we know a solution V and W exist, the set of linear equations will have a solution. But it could also have many solutions. However, any solution we obtain will satisfy $V(x)/W(x) = P(x)$. To see this, let us suppose we have two pairs of solutions (V_1, W_1) and (V_2, W_2). Clearly $V_1(i)W_2(i)f_i = V_2(i)W_1(i)f_i$. If f_i does not equal 0, then $V_1(i)W_2(i) = V_2(i)W_1(i)$ by cancellation. But if f_i does equal 0, then $V_1(i)W_2(i) = V_2(i)W_1(i)$ since then $V_1(i) = V_2(i) = 0$. But this means that the polynomials V_1W_2 and V_2W_1 must be equal, since they agree on n points and each has degree $d + 2k - 1 < n$. But if these polynomials are equal, then $V_1(x)/W_1(x) = V_2(x)/W_2(x)$. Since any solution (V,W) yields the same ratio $V(x)/W(x)$, this ratio must always equal $P(x)$!

Arithmetic in $GF(2^r)$

In practice, we want our Reed-Solomon codes to be very efficient. In this regard, working in $GF(p)$ for some prime is inconvenient, for several reasons. Let us suppose it is most convenient if we work in blocks of 8 bits. If we work in $GF(251)$, we are not using all the possibilities for our eight bits. Besides being wasteful, this is problematic if our data (which may come from text, compressed data, etc.) contains a block of eight bits which corresponds to the number 252!

It is therefore more natural to work in a field with 2^r elements, or $GF(2^r)$. Arithmetic is this field is done by finding an irreducible (prime) polynomial $\pi(x)$ of degree r, and doing all arithmetic in $Z_2[\pi(x)]$. That is, all coefficients are modulo 2, arithmetic is done modulo $\pi(x)$, and $\pi(x)$ should not be able to be factored over $GF(2)$.

For example, for $GF(2^8)$, an irreducible polynomial is $\pi(x) = x^8 + x^6 + x^5 + x + 1$. A byte can naturally be though of as a polynomial in the field. For example, by letting the least significant bit represent x^0, and the ith least significant bit represent x^i, we have that the byte 10010010 represents the polynomial $x^7 + x^4 + x$. Adding in $GF(2^r)$ is easy: since all coefficients are modulo 2, we can just XOR two bytes together. For example

$$\begin{aligned} 10010010 + 10101010 &= 00111000 \\ (x^7 + x^4 + x) + (x^7 + x^5 + x^3 + x) &= x^5 + x^4 + x^3. \end{aligned}$$

Moreover, subtracting is just the same as adding!

Multiplication is slightly harder, since we work modulo $\pi(x)$. As an example

$$(x^4 + x) \cdot (x^4 + x^2) = x^8 + x^6 + x^5 + x^3.$$

However, we must reduce this so that we can fit it into a byte. As we work modulo $\pi(x)$, we have that $\pi(x) = 0$, or $x^8 = x^6 + x^5 + x + 1$. Hence

$$(x^4 + x) \cdot (x^4 + x^2) = x^8 + x^6 + x^5 + x^3 = (x^6 + x^5 + x + 1) + x^6 + x^5 + x^3 = x^3 + x + 1,$$

and hence

$$00010010 \cdot 00010100 = 00001011.$$

Rather than compute these products on the fly, all possible $256 \cdot 256$ pairs can be precomputed once in the beginning, and then all multiplications are done by just doing a lookup in the multiplication lookup table. Hence by using memory and preprocessing, one can work in $GF(2^8)$ and still obtain great speed.

Reed-Solomon codes work exactly the same over $GF(2^r)$ as they do over $GF(p)$, since in both cases the main reqiurement, namely that a polynomial of degree $d-1$ be uniquely defined by d points, is satisfied.

Reed-Solomon Codes

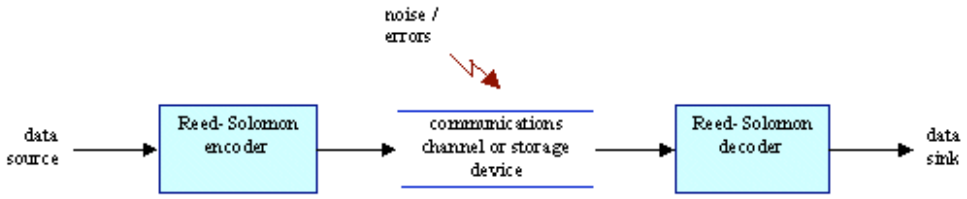
An introduction to Reed-Solomon codes: principles, architecture and implementation

1. Introduction

Reed-Solomon codes are block-based error correcting codes with a wide range of applications in digital communications and storage. Reed-Solomon codes are used to correct errors in many systems including:

- Storage devices (including tape, Compact Disk, DVD, barcodes, etc)
- Wireless or mobile communications (including cellular telephones, microwave links, etc)
- Satellite communications
- Digital television / DVB
- High-speed modems such as ADSL, xDSL, etc.

A typical system is shown here:



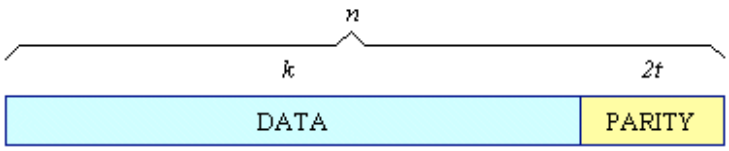
The Reed-Solomon encoder takes a block of digital data and adds extra "redundant" bits. Errors occur during transmission or storage for a number of reasons (for example noise or interference, scratches on a CD, etc). The Reed-Solomon decoder processes each block and attempts to correct errors and recover the original data. The number and type of errors that can be corrected depends on the characteristics of the Reed-Solomon code.

2. Properties of Reed-Solomon codes

Reed Solomon codes are a subset of BCH codes and are linear block codes. A Reed-Solomon code is specified as RS(n,k) with s-bit symbols.

This means that the encoder takes k data symbols of s bits each and adds parity symbols to make an n symbol codeword. There are n-k parity symbols of s bits each. A Reed-Solomon decoder can correct up to t symbols that contain errors in a codeword, where $2t = n-k$.

The following diagram shows a typical Reed-Solomon codeword (this is known as a Systematic code because the data is left unchanged and the parity symbols are appended):



Example: A popular Reed-Solomon code is RS(255,223) with 8-bit symbols. Each codeword contains 255 code word bytes, of which 223 bytes are data and 32 bytes are parity. For this code:

n = 255, k = 223, s = 8

2t = 32, t = 16

The decoder can correct any 16 symbol errors in the code word: i.e. errors in up to 16 bytes anywhere in the codeword can be automatically corrected.

Given a symbol size s, the maximum codeword length (n) for a Reed-Solomon code is $\mathbf{n = 2^s - 1}$

For example, the maximum length of a code with 8-bit symbols (s=8) is 255 bytes.

Reed-Solomon codes may be shortened by (conceptually) making a number of data symbols zero at the encoder, not transmitting them, and then re-inserting them at the decoder.

Example: The (255,223) code described above can be shortened to (200,168). The encoder takes a block of 168 data bytes, (conceptually) adds 55 zero bytes, creates a (255,223) codeword and transmits only the 168 data bytes and 32 parity bytes.

The amount of processing "power" required to encode and decode Reed-Solomon codes is related to the number of parity symbols per codeword. A large value of t means that a large number of errors can be corrected but requires more computational power than a small value of t.

Symbol Errors

One symbol error occurs when 1 bit in a symbol is wrong or when all the bits in a symbol are wrong.

Example: RS(255,223) can correct 16 symbol errors. In the worst case, 16 bit errors may occur, each in a separate symbol (byte) so that the decoder corrects 16 bit errors. In the best case, 16 complete byte errors occur so that the decoder corrects 16 x 8 bit errors.

Reed-Solomon codes are particularly well suited to correcting burst errors (where a series of bits in the codeword are received in error).

Decoding

Reed-Solomon algebraic decoding procedures can correct errors and erasures. An erasure occurs when the position of an erred symbol is known. A decoder can correct up to *t* errors or up to *2t* erasures. Erasure information can often be supplied by the demodulator in a digital communication system, i.e. the demodulator "flags" received symbols that are likely to contain errors.

When a codeword is decoded, there are three possible outcomes:

1. If 2s + r < 2t (s errors, r erasures) then the original transmitted code word will always be recovered,

OTHERWISE

2. The decoder will detect that it cannot recover the original code word and indicate this fact.

OR

3. The decoder will mis-decode and recover an incorrect code word without any indication.

The probability of each of the three possibilities depends on the particular Reed-Solomon code and on the

number and distribution of errors.

Coding Gain

The advantage of using Reed-Solomon codes is that the probability of an error remaining in the decoded data is (usually) much lower than the probability of an error if Reed-Solomon is not used. This is often described as **coding gain**.

> Example: A digital communication system is designed to operate at a Bit Error Ratio (BER) of 10^{-9}, i.e. no more than 1 in 10^9 bits are received in error. This can be achieved by boosting the power of the transmitter **or** by adding Reed-Solomon (or another type of Forward Error Correction). Reed-Solomon allows the system to achieve this target BER with a lower transmitter output power. The power "saving" given by Reed-Solomon (in decibels) is the **coding gain**.

3. Architectures for encoding and decoding Reed-Solomon codes

Reed-Solomon encoding and decoding can be carried out in software or in special-purpose hardware.

Finite (Galois) Field Arithmetic

Reed-Solomon codes are based on a specialist area of mathematics known as Galois fields or finite fields. A finite field has the property that arithmetic operations (+,-,x,/ etc.) on field elements always have a result in the field. A Reed-Solomon encoder or decoder needs to carry out these arithmetic operations. These operations require special hardware or software functions to implement.

Generator Polynomial

A Reed-Solomon codeword is generated using a special polynomial. All valid codewords are exactly divisible by the generator polynomial. The general form of the generator polynomial is:

$$g(x) = (x - \alpha^i)(x - \alpha^{i+1})\dots(x - \alpha^{i+2t})$$

and the codeword is constructed using:

$$c(x) = g(x).i(x)$$

where g(x) is the generator polynomial, i(x) is the information block, c(x) is a valid codeword and a is referred to as a primitive element of the field.

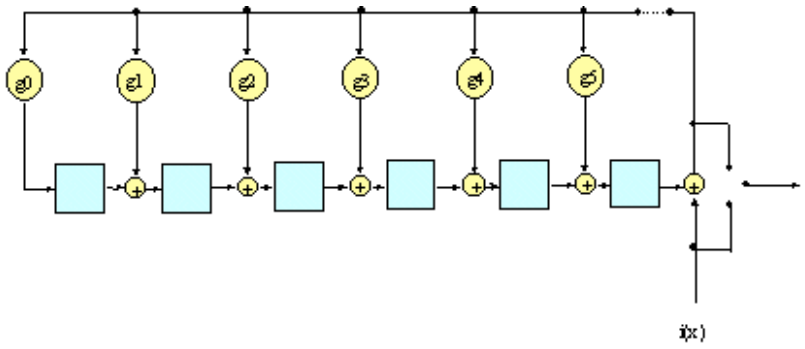
> Example: Generator for RS(255,249)
>
> $$g(x) = (x - \alpha^0)(x - \alpha^1)(x - \alpha^2)(x - \alpha^3)(x - \alpha^4)(x - \alpha^5)$$
> $$g(x) = x^6 + g_5x^5 + g_4x^4 + g_3x^3 + g_2x^2 + g_1x^1 + g_0$$

3.1 Encoder architecture

The *2t* parity symbols in a systematic Reed-Solomon codeword are given by:

$$p(x) = i(x) \cdot x^{n-k} \bmod g(x)$$

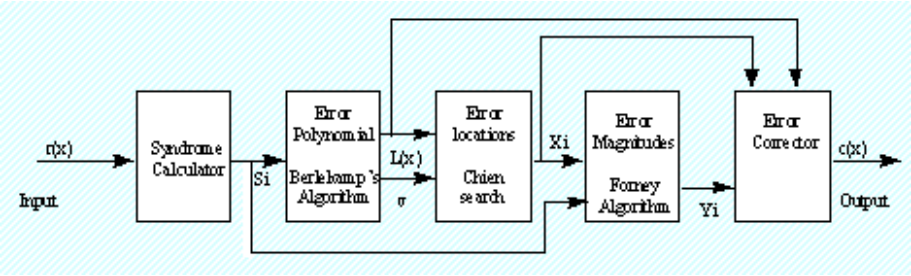
The following diagram shows an architecture for a systematic RS(255,249) encoder:



Each of the 6 registers holds a symbol (8 bits). The arithmetic operators carry out finite field addition or multiplication on a complete symbol.

3.2 Decoder architecture

A general architecture for decoding Reed-Solomon codes is shown in the following diagram.



Key

r(x)	Received codeword
Si	Syndromes
L(x)	Error locator polynomial
Xi	Error locations
Yi	Error magnitudes
c(x)	Recovered code word
v	Number of errors

The received codeword r(x) is the original (transmitted) codeword c(x) plus errors:

$$r(x) = c(x) + e(x)$$

A Reed-Solomon decoder attempts to identify the position and magnitude of up to t errors (or 2t erasures) and to correct the errors or erasures.

Syndrome Calculation

This is a similar calculation to parity calculation. A Reed-Solomon codeword has 2t **syndromes** that depend only on errors (not on the transmitted code word). The syndromes can be calculated by substituting the 2t roots of the generator polynomial g(x) into r(x).

Finding the Symbol Error Locations

This involves solving simultaneous equations with t unknowns. Several fast algorithms are available to do this. These algorithms take advantage of the special matrix structure of Reed-Solomon codes and greatly reduce the computational effort required. In general two steps are involved:

Find an error locator polynomial

This can be done using the Berlekamp-Massey algorithm or Euclid's algorithm. Euclid's algorithm tends to be more widely used in practice because it is easier to implement: however, the Berlekamp-Massey algorithm tends to lead to more efficient hardware and software implementations.

Find the roots of this polynomial

This is done using the Chien search algorithm.

Finding the Symbol Error Values

Again, this involves solving simultaneous equations with t unknowns. A widely-used fast algorithm is the Forney algorithm.

4. Implementation of Reed-Solomon encoders and decoders

Hardware Implementation

A number of commercial hardware implementations exist. Many existing systems use "off-the-shelf" integrated circuits that encode and decode Reed-Solomon codes. These ICs tend to support a certain amount of programmability (for example, RS(255,k) where t = 1 to 16 symbols). A recent trend is towards VHDL or Verilog designs (**logic cores** or **intellectual property cores**). These have a number of advantages over standard ICs. A logic core can be integrated with other VHDL or Verilog components and synthesized to an FPGA (Field Programmable Gate Array) or ASIC (Application Specific Integrated Circuit) – this enables so-called "System on Chip" designs where multiple modules can be combined in a single IC. Depending on production volumes, logic cores can often give significantly lower system costs than "standard" ICs. By using logic cores, a designer avoids the potential need to do a "lifetime buy" of a Reed-Solomon IC.

Software Implementation

Until recently, software implementations in "real-time" required too much computational power for all but the simplest of Reed-Solomon codes (i.e. codes with small values of t). The major difficulty in implementing Reed-Solomon codes in software is that general purpose processors do not support Galois field arithmetic operations. For example, to implement a Galois field multiply in software requires a test for 0, two log table look-ups, modulo add and anti-log table look-up. However, careful design together with increases in processor performance mean that software implementations can operate at relatively high data rates. The following table gives some example benchmark figures on a 166MHz Pentium PC:

Code	Data rate
RS(255,251)	12 Mbps
RS(255,239)	2.7 Mbps

RS(255,223)	1.1 Mbps

These data rates are for decoding only: encoding is considerably faster since it requires less computation.

5. Further reading

In this paper we have deliberately avoided discussing the theory and implementation of Reed-Solomon codes in detail. For more detail please see the following books:

1.Wicker, "Error Control Systems for Digital Communication and Storage", Prentice-Hall 1995

2. Lin and Costello, "Error Control Coding: Fundamentals and Applications", Prentice-Hall 1983

3. Clark and Cain, "Error Correction Coding for Digital Communications", Plenum 1988

4. Wilson, "Digital Modulation and Coding", Prentice-Hall 1996

6. About the authors

This paper was written by Martyn Riley and Iain Richardson. For more details about the authors click here.