Reed–Solomon error correction

From Wikipedia, the free encyclopedia

Reed–Solomon codes are an important group of error-correcting codes that were introduced by Irving S. Reed and Gustave Solomon in 1960.[1] They have many important applications, the most prominent of which include consumer technologies such as CDs, DVDs, Blu-ray Discs, QR Codes, data transmission technologies such as DSL and WiMAX, broadcast systems such as DVB and ATSC, and storage systems such as RAID 6. They are also used in satellite communication.

In coding theory, the Reed–Solomon code belongs to the class of non-binary cyclic error-correcting codes. The Reed–Solomon code is based on univariate polynomials over finite fields.

It is able to detect and correct multiple symbol errors. By adding t check symbols to the data, a Reed–Solomon code can detect any combination of up to t erroneous symbols, or correct up to $\lfloor t/2 \rfloor$ symbols. As an erasure code, it can correct up to t known erasures, or it can detect and correct combinations of errors and erasures. Furthermore, Reed–Solomon codes are suitable as multiple-burst bit-error correcting codes, since a sequence of $b + 1$ consecutive bit errors can affect at most two symbols of size b. The choice of t is up to the designer of the code, and may be selected within wide limits.

Reed–Solomon codes

Named after	Irving S. Reed and Gustave Solomon
Classification	
Hierarchy	Linear block code Polynomial code Cyclic code BCH code Reed–Solomon code
Parameters	
Block length	n
Message length	k
Distance	$n - k + 1$
Alphabet size	$q = p^m \geq n$ (p prime) Often $n = q - 1$.
Notation	$[n, k, n - k + 1]_q$-code
Algorithms	
Decoding	Berlekamp–Massey Euclidean *et al.*
Properties	
Maximum-distance separable code	

v · t · e

Contents [hide]

History [edit]

Reed–Solomon codes were developed in 1960 by Irving S. Reed and Gustave Solomon, who were then staff members of MIT Lincoln Laboratory. Their seminal article was entitled "Polynomial Codes over Certain Finite Fields." (Reed & Solomon 1960). When the article was written, an efficient decoding algorithm was not known. A solution for the latter was found in 1969 by Elwyn Berlekamp and James Massey, and is since known as the Berlekamp–Massey decoding algorithm. In 1977, Reed–Solomon codes were notably implemented in the Voyager program in the form of concatenated codes. The first commercial application in mass-produced consumer products appeared in 1982 with the compact disc, where two interleaved Reed–Solomon codes are used. Today, Reed–Solomon codes are widely implemented in digital storage devices and digital communication standards, though they are being slowly replaced by more modern low-density parity-check (LDPC) codes or turbo codes. For example, Reed–Solomon codes are used in the digital video broadcasting (DVB) standard DVB-S, but LDPC codes are used in its successor DVB-S2.

Basis [edit]

The Reed–Solomon code is based on univariate polynomials over finite fields; in particular, for some parameters k and n, the codewords of the Reed–Solomon code consists of all function tables of polynomials of degree less than k over the finite field with n elements - for this to work, n has to be prime power. The encoding scheme of the Reed–Solomon code turns k symbols into such a function table, which is essentially a list of n symbols. One way to perform this encoding is by interpreting the k given symbols as the first segment of the function table of a polynomial of degree less than k. A simple argument shows that there is exactly one such polynomial, and the remaining $n - k$ symbols can thus be generated by evaluating the polynomial at those points. Since the n transmitted symbols form an overdetermined system that specifies a polynomial of degree less than k, we can use interpolation techniques at the receiver to recover the original message in case not too many errors happened.

In order to achieve the most efficient decoding procedures, the encoding procedure of the Reed–Solomon code is often constructed a bit differently, namely as cyclic BCH codes: the k source symbols are interpreted as the *coefficients* of a polynomial of degree less than k, and the additional $n - k$ symbols are derived from the coefficients of a polynomial constructed by multiplying $p(x)$ with a cyclic generator polynomial.

Applications [edit]

Data storage [edit]

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.[2]

DVDs use a similar scheme, but with much larger blocks, a (208,192) inner code, and a (182,172) outer code.

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.

Bar code [edit]

Almost all two-dimensional bar codes such as PDF-417, MaxiCode, Datamatrix, QR Code, and Aztec Code use Reed–Solomon error correction to allow correct reading even if a portion of the bar code is damaged. When the bar code scanner cannot recognize a bar code symbol, it will treat it as an erasure.

Reed–Solomon coding is less common in one-dimensional bar codes, but is used by the PostBar symbology.

Data transmission [edit]

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 2*K*, 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.

Space transmission [edit]

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.

Constructions [edit]

The Reed–Solomon code is actually a family of codes: For every choice of the three parameters *k* < *n* ≤ *q*, there is a Reed–Solomon code that has an alphabet of size *q*, a block length *n*, and a message length *k*. Moreover, the alphabet is interpreted as the finite field of order *q*, and thus, *q* has to be a prime power. In the most useful parameterizations of the Reed–Solomon code, the block length is usually some constant multiple of the message length, that is, the rate *R* = *k*/*n* is some constant, and furthermore, the block length is equal to or one less than the alphabet size, that is, *n* = *q* or *n* = *q*−1.

Reed & Solomon's original view: The codeword as a sequence of values [edit]

There are different encoding procedures for the Reed–Solomon code, and thus, there are different ways to describe the set of all codewords. In the original view of Reed & Solomon (1960), every codeword of the Reed–Solomon code is a sequence of function values of a low-degree polynomial. More precisely, in order to obtain a codeword of the Reed–Solomon code, the message is interpreted as the description of a polynomial *p* of degree less than *k* over the finite field *F* with *q* elements. In turn, the polynomial *p* is evaluated at *n* distinct points $a_1, \dots, a_n$ of the field *F*, and the sequence of values is the corresponding codeword. Formally, the set $\mathbf{C}$ of codewords of the Reed–Solomon code is defined as follows:

$$\mathbf{C} = \Big\{\; \big(p(a_1), p(a_2), \dots, p(a_n)\big) \;\Big|\; p \text{ is a polynomial over } F \text{ of degree } < k \;\Big\}.$$

Since any two *different* polynomials of degree less than k agree in at most $k-1$ points, this means that any two codewords of the Reed–Solomon code disagree in at least $n-(k-1)=n-k+1$ positions. Furthermore, there are two polynomials that do agree in $k-1$ points but are not equal, and thus, the distance of the Reed–Solomon code is exactly $d=n-k+1$. Then the relative distance is $\delta = d/n = 1-k/n+1/n \sim 1-R$, where $R=k/n$ is the rate. This trade-off between the relative distance and the rate is asymptotically optimal since, by the Singleton bound, *every* code satisfies $\delta + R \le 1$. Being a code that achieves this optimal trade-off, the Reed–Solomon code belongs to the class of maximum distance separable codes.

While the number of different polynomials of degree less than *k* and the number of different messages are both equal to q^k, and thus every message can be uniquely mapped to such a polynomial, there are different ways of doing this encoding. The original construction of Reed & Solomon (1960) interprets the message *x* as the *coefficients* of the polynomial *p*, whereas subsequent constructions interpret the message as the *values* of the polynomial at the first *k* points $a_1, \dots, a_k$ and obtain the polynomial *p* by interpolating these values with a polynomial of degree less than *k*. The latter encoding procedure, while being slightly less efficient, has the advantage that it gives rise to a systematic code, that is, the original message is always contained as a subsequence of the codeword.

In many contexts it is convenient to choose the sequence $a_1, \dots, a_n$ of evaluation points so that they exhibit some additional structure. In particular, it is useful to choose the sequence of successive powers of a primitive root α of the field F, that is, α is generator of the finite field's multiplicative group and the sequence is defined as $a_i = \alpha^i$ for all $i = 1, \dots, q-1$. This sequence contains all elements of F except for 0, so in this setting, the block length is $n = q-1$. Then it follows that, whenever $p(a)$ is a polynomial over F, then the function $p(\alpha a)$ is also a polynomial of the same degree, which gives rise to a codeword that is a cyclic left-shift of the codeword derived from $p(a)$; thus, this construction of Reed–Solomon codes gives rise to a cyclic code.

Simple encoding procedure: The message as a sequence of coefficients [edit]

In the original construction of Reed & Solomon (1960), the message $x = (x_1, \dots, x_k) \in F^k$ is mapped to the polynomial p_x with

$$p_x(a) = \sum_{i=1}^{k} x_i a^{i-1}\,.$$

As described above, the codeword is then obtained by evaluating p at n different points $a_1, \dots, a_n$ of the field F. Thus the classical encoding function $C: F^k \to F^n$ for the Reed–Solomon code is defined as follows:

$$C(x) = \big(p_x(a_1), \dots, p_x(a_n)\big)\,.$$

This function C is a linear mapping, that is, it satisfies $C(x) = x \cdot A$ for the following $(k \times n)$-matrix A with elements from F:

$$A = \begin{bmatrix} 1 & \dots & 1 \\ a_1 & \dots & a_n \\ a_1^2 & \dots & a_n^2 \\ \vdots & \ddots & \vdots \\ a_1^{k-1} & \dots & a_n^{k-1} \end{bmatrix}$$

This matrix is the transpose of a Vandermonde matrix over F. In other words, the Reed–Solomon code is a linear code, and in the classical encoding procedure, its generator matrix is A.

Systematic encoding procedure: The message as an initial sequence of values [edit]

As mentioned above, there is an alternative way to map codewords x to polynomials p_x. In this alternative encoding procedure, the polynomial p_x is the unique polynomial of degree less than k such that

$p_x(a_i) = x_i$ holds for all $i = 1, \dots, k$.

To compute this polynomial p_x from x, one can use Lagrange interpolation. Once it has been found, it is evaluated at the other points $a_{k+1}, \dots, a_n$ of the field. The alternative encoding function $C: F^k \to F^n$ for the Reed–Solomon code is then again just the sequence of values:

$$C(x) = \big(p_x(a_1), \dots, p_x(a_n)\big)\,.$$

This time, however, the first k entries of the codeword are exactly equal to x, so this encoding procedure gives rise to a systematic code. It can be checked that the alternative encoding function is a linear mapping as well.

The BCH view: The codeword as a sequence of coefficients [edit]

In this view, the sender again maps the message x to a polynomial p_x, and for this, any of the two mappings above can be used (where the message is either interpreted as the coefficients of p_x or as the initial sequence of values of p_x). Once the sender has constructed the polynomial p_x in some way, however, instead of sending the *values* of p_x at all points, the sender computes some related polynomial s of degree at most $n-1$ for $n=q-1$ and sends the n *coefficients* of that polynomial. The polynomial $s(a)$ is constructed by multiplying the message polynomial $p_x(a)$, which has degree at most $k-1$, with a generator polynomial $g(a)$ of degree $n-k$ that is known to both the sender and the receiver.[3] The generator polynomial $g(x)$ is defined as the polynomial whose roots are exactly $\alpha, \alpha^2, \dots, \alpha^{n-k}$, i.e.,

$$g(x) = (x-\alpha)(x-\alpha^2)\cdots(x-\alpha^{n-k}) = g_0 + g_1x + \cdots + g_{n-k-1}x^{n-k-1} + x^{n-k}\,.$$

The transmitter sends the $n=q-1$ coefficients of $s(a)=p_x(a)\cdot g(a)$. Thus, in the BCH view of Reed Solomon codes, the set $\mathbf{C}'$ of codewords is defined for $n=q-1$ as follows:[4]

$$\mathbf{C}' = \left\{ (s_1, s_2, \dots, s_n) \;\middle|\; s(a) = \sum_{i=1}^{n} s_i a^{i-1} \text{ is a polynomial that has at least the roots } \alpha^1, \alpha^2, \dots, \alpha^{n-k} \right\}.$$

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

Since Reed–Solomon codes are a special case of BCH codes and the Berlekamp–Massey algorithm has been designed for the decoding of such codes, it is applicable to Reed–Solomon codes: The receiver interprets the received word as the coefficients of a polynomial $r(a)$. If no error has occurred during the transmission, that is, if $r(a)=s(a)$, then the receiver can use polynomial division to determine the message polynomial $p_x(a)=r(a)/g(a)$. In general, the receiver can use polynomial division to construct the unique polynomials $p(a)$ and $e(a)$, such that $e(a)$ has degree less than the degree of $g(a)$ and

$$r(a) = p(a)\cdot g(a) + e(a)\,.$$

If the remainder polynomial $e(a)$ is not identically zero, then an error has occurred during the transmission. The receiver can evaluate $r(a)$ at the roots of $g(a)$ and build a system of equations that eliminates $s(a)$ and identifies which coefficients of $r(a)$ are in error, and the magnitude of each coefficient's error (Berlekamp 1984 and Massey 1969). If the system of equations can be solved, then the receiver knows how to modify the received word $r(a)$ to get the most likely codeword $s(a)$ that was sent.

Systematic encoding procedure [edit]

The above encoding procedure for the BCH view of Reed–Solomon codes is classical, but does not give rise to a systematic encoding procedure, i.e., the codewords do not necessarily contain the message as a subsequence. To remedy this fact, instead of sending $s(x)=p(x)g(x)$, the encoder constructs the transmitted polynomial $s(x)$ such that the coefficients of the k largest monomials are equal to the corresponding coefficients of $p(x)$, and the lower-order coefficients of $s(x)$ are chosen exactly in such a way that $s(x)$ becomes evenly divisible by $g(x)$. Then the coefficients of $p(x)$ are a subsequence of the coefficients of $s(x)$. To get a code that is overall systematic, we construct the message polynomial $p(x)$ by interpreting the message as the sequence of its coefficients.

Formally, the construction is done by multiplying $p(x)$ by x^t to make room for the $t=n-k$ check symbols, dividing that product by $g(x)$ to find the remainder, and then compensating for that remainder by subtracting it. The t check symbols are created by computing the remainder $s_r(x)$:

$$s_r(x) = p(x)\cdot x^t \mod g(x)\,.$$

Note that the remainder has degree at most $t-1$, whereas the coefficients of $x^{t-1}, x^{t-2}, \dots, x^1, x^0$ in the polynomial $p(x)\cdot x^t$ are zero. Therefore, the following definition of the codeword $s(x)$ has the property that the first k coefficients are identical to the coefficients of $p(x)$:

$$s(x) = p(x)\cdot x^t - s_r(x)\,.$$

As a result, the codewords $s(x)$ are indeed elements of $\mathbf{C}'$, that is, they are evenly divisible by the generator polynomial $g(x)$:[5]

$$s(x) \equiv p(x)\cdot x^t - s_r(x) \equiv s_r(x) - s_r(x) \equiv 0 \mod g(x)\,.$$

Equivalence of the two views [edit]

At first sight, the two views of Reed–Solomon codes above seem very different. In the first definition, codewords in the set $\mathbf{C}$ are *values* of polynomials, whereas in the second set $\mathbf{C}'$, 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. Yet, it can be shown that the two sets are actually equal, that is, $\mathbf{C}=\mathbf{C}'$ holds (for an appropriate choice of $a_1, \dots, a_n$).

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 $i=0,\dots,n-1$, where α is a primitive nth root of unity.

To be more precise, let

$$p(x) = v_0 + v_1x + v_2x^2 + \cdots + v_{n-1}x^{n-1},$$
$$q(x) = f_0 + f_1x + f_2x^2 + \cdots + f_{n-1}x^{n-1}$$

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 $i=0,\dots,n-1$, $f_i=p(\alpha^i)$ 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 $i = k, \ldots, n-1$,
- if and only if $q(\alpha^i) = 0$ for $i = 1, \ldots, n-k$ (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.

Remarks [edit]

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 source block 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[6] theorem illustrates an example of an application of shortened Reed–Solomon codes. In parallel to shortening, a technique known as puncturing allows omitting some of the encoded parity symbols.

Properties [edit]

The Reed–Solomon code is a [*n*, *k*, *n* − *k* + 1] code; in other words, it is a linear block code of length *n* (over *F*) with dimension *k* and minimum Hamming distance *n* − *k* + 1. The Reed–Solomon code is optimal in the sense that the minimum distance has the maximum value possible for a linear code of size (*n*, *k*); this is known as the Singleton bound. Such a code is also called a maximum distance separable (MDS) code.

The error-correcting ability of a Reed–Solomon code is determined by its minimum distance, or equivalently, by $n-k$, the measure of redundancy in the block. If the locations of the error 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 2*E* + *S* ≤ *n* − *k* is satisfied, where E is the number of errors and S is the number of erasures in the block.

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 Reed–Solomon code properties discussed above 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 compared to a binary code.

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 inversion of the original data. However, the Reed–Solomon code loses its transparency when the code is shortened. 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 one-fill.) For this reason it is mandatory that the sense of the data (i.e., true or complemented) be resolved before Reed–Solomon decoding.

Error correction algorithms [edit]

Theoretical decoder [edit]

Reed & Solomon (1960) described a theoretical decoder that corrected errors by finding the most popular message polynomial. The decoder for a RS (n, k) code would look at all possible subsets of k symbols from the set of n symbols that were received. For the code to be correctable in general, at least k symbols had to be received correctly, and k symbols are needed to interpolate the message polynomial. The decoder would interpolate a message polynomial for each subset, and it would keep track of the resulting polynomial candidates. The most popular message is the corrected result. Unfortunately, there are a lot of subsets, so the algorithm is impractical. The number of subsets is the binomial coefficient, $\binom{n}{k} = \frac{n!}{(n-k)!k!}$, and the number of subsets is infeasible for even modest codes. For a $(255, 249)$ code that can correct 3 errors, the naive theoretical decoder would examine 359 billion subsets. The Reed–Solomon code needed a practical decoder.

Peterson decoder [edit]

Main article: Peterson–Gorenstein–Zierler algorithm

Peterson (1960) developed a practical decoder based on syndrome decoding. (Welch 1997, p. 10) Berlekamp (below) would improve on that decoder.

Syndrome decoding [edit]

The transmitted message is viewed as the coefficients of a polynomial *s*(*x*) that is divisible by a generator polynomial *g*(*x*). Welch (1997, p. 5)

$$s(x) = \sum_{i=0}^{n-1} c_i x^i$$

$$g(x) = \prod_{j=1}^{n-k} (x - \alpha^j),$$

where *α* is a primitive root.

Since *s*(*x*) is divisible by generator *g*(*x*), it follows that

$$s(\alpha^i) = 0,\ i = 1, 2, \ldots, n-k$$

The transmitted polynomial is corrupted in transit by an error polynomial *e*(*x*) to produce the received polynomial *r*(*x*).

$$r(x) = s(x) + e(x)$$
$$e(x) = \sum_{i=0}^{n-1} e_i x^i$$

where e_i is the coefficient for the *i*-th power of *x*. Coefficient e_i will be zero if there is no error at that power of *x* and nonzero if there is an error. If there are *ν* errors at distinct powers i_k of *x*, then

$$e(x) = \sum_{k=1}^{\nu} e_{i_k} x^{i_k}$$

The goal of the decoder is to find the number of errors (*ν*), the positions of the errors (i_k), and the error values at those positions (e_{i_k}). From those, e(x) can be calculated and subtracted from r(x) to get the original message s(x).

The syndromes S_j are defined as

$$S_j = r(\alpha^j) = s(\alpha^j) + e(\alpha^j) = 0 + e(\alpha^j) = e(\alpha^j),\ j = 1, 2, \ldots, n-k$$
$$= \sum_{k=1}^{\nu} e_{i_k} \left(\alpha^j\right)^{i_k}$$

The advantage of looking at the syndromes is that the message polynomial drops out. Another possible way of calculating e(x) is using polynomial interpolation to find the only polynomial that passes through the points (α^j, S_j) (Because $S_j = e(\alpha^j)$), however, this is not used widely because polynomial interpolation is not always feasible in the fields used in Reed–Solomon error correction. For example, it is feasible over the integers (of course), but it is infeasible over the integers modulo a prime[*citation needed*].

Error locators and error values [edit]

For convenience, define the **error locators** X_k and **error values** Y_k as:

$$X_k = \alpha^{i_k},\ Y_k = e_{i_k}$$

Then the syndromes can be written in terms of the error locators and error values as

$$S_j = \sum_{k=1}^{\nu} Y_k X_k^j$$

The syndromes give a system of *n* − *k* ≥ 2*ν* equations in 2*ν* unknowns, but that system of equations is nonlinear in the X_k and does not have an obvious solution. However, if the X_k were known (see below), then the syndrome equations provide a linear system of equations that can easily be solved for the Y_k error values.

$$\begin{bmatrix} X_1^1 & X_2^1 & \cdots & X_\nu^1 \\ X_1^2 & X_2^2 & \cdots & X_\nu^2 \\ \vdots & \vdots & & \vdots \\ X_1^{n-k} & X_2^{n-k} & \cdots & X_\nu^{n-k} \end{bmatrix} \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_\nu \end{bmatrix} = \begin{bmatrix} S_1 \\ S_2 \\ \vdots \\ S_{n-k} \end{bmatrix}$$

Consequently, the problem is finding the X_k, because then the leftmost matrix would be known, and both sides of the equation could be multiplied by its inverse, yielding Y_k

Error locator polynomial [edit]

Peterson found a linear recurrence relation that gave rise to a system of linear equations. (Welch 1997, p. 10) Solving those equations identifies the error locations.

Define the **error locator polynomial** Λ(*x*) as

$$\Lambda(x) = \prod_{k=1}^{\nu} (1 - x X_k) = 1 + \Lambda_1 x^1 + \Lambda_2 x^2 + \cdots + \Lambda_\nu x^\nu$$

The zeros of Λ(*x*) are the reciprocals X_k^{-1}:

$$\Lambda(X_k^{-1}) = 0$$
$$\Lambda(X_k^{-1}) = 1 + \Lambda_1 X_k^{-1} + \Lambda_2 X_k^{-2} + \cdots + \Lambda_\nu X_k^{-\nu} = 0$$

Multiply both sides by $Y_k X_k^{j+\nu}$ and it will still be zero. j is any number such that 1≤j≤v.

$$Y_k X_k^{j+\nu} \Lambda(X_k^{-1}) = 0.$$
$$\text{Hence } Y_k X_k^{j+\nu} + \Lambda_1 Y_k X_k^{j+\nu} X_k^{-1} + \Lambda_2 Y_k X_k^{j+\nu} X_k^{-2} + \cdots + \Lambda_\nu Y_k X_k^{j+\nu} X_k^{-\nu} = 0,$$
$$\text{and so } Y_k X_k^{j+\nu} + \Lambda_1 Y_k X_k^{j+\nu-1} + \Lambda_2 Y_k X_k^{j+\nu-2} + \cdots + \Lambda_\nu Y_k X_k^{j} = 0$$

Sum for *k* = 1 to *ν*

$$\sum_{k=1}^{\nu} (Y_k X_k^{j+\nu} + \Lambda_1 Y_k X_k^{j+\nu-1} + \Lambda_2 Y_k X_k^{j+\nu-2} + \cdots + \Lambda_\nu Y_k X_k^{j}) = 0$$
$$\sum_{k=1}^{\nu} (Y_k X_k^{j+\nu}) + \Lambda_1 \sum_{k=1}^{\nu} (Y_k X_k^{j+\nu-1}) + \Lambda_2 \sum_{k=1}^{\nu} (Y_k X_k^{j+\nu-2}) + \cdots + \Lambda_\nu \sum_{k=1}^{\nu} (Y_k X_k^{j}) = 0$$

This reduces to

$$S_{j+\nu} + \Lambda_1 S_{j+\nu-1} + \cdots + \Lambda_{\nu-1} S_{j+1} + \Lambda_\nu S_j = 0$$
$$S_j \Lambda_\nu + S_{j+1}\Lambda_{\nu-1} + \cdots + S_{j+\nu-1}\Lambda_1 = -S_{j+\nu}$$

This yields a system of linear equations that can be solved for the coefficients Λ_i of the error location polynomial:

$$\begin{bmatrix} S_1 & S_2 & \cdots & S_\nu \\ S_2 & S_3 & \cdots & S_{\nu+1} \\ \vdots & \vdots & & \vdots \\ S_\nu & S_{\nu+1} & \cdots & S_{2\nu-1} \end{bmatrix} \begin{bmatrix} \Lambda_\nu \\ \Lambda_{\nu-1} \\ \vdots \\ \Lambda_1 \end{bmatrix} = \begin{bmatrix} -S_{\nu+1} \\ -S_{\nu+2} \\ \vdots \\ -S_{\nu+\nu} \end{bmatrix}$$

The above assumes the decoder knows the number of errors *v*, but that number has not been determined yet. The PGZ decoder does not determine *v* directly but rather searches for it by trying successive values. The decoder first assumes the largest value for a trial *v* and sets up the linear system for that value. If the equations can be solved (i.e., the matrix determinant is nonzero), then that trial value is the number of errors. If the linear system cannot be solved, then the trial *v* is reduced by one and the next smaller system is examined. (Gill unknown, p. 35)

Obtain the error locators from the error locator polynomial [edit]

Use the coefficients Λ_i found in the last step to build the error location polynomial. The roots of the error location polynomial can be found by exhaustive search. The error locators are the reciprocals of those roots. Chien search is an efficient implementation of this step.

Calculate the error locations [edit]

Calculate i_k by taking the log base a of X_k. This is generally done using a precomputed lookup table.

Calculate the error values [edit]

Once the error locators are known, the error values can be determined. This can be done by direct solution for Y_k in the error equations given above, or using the Forney algorithm.

Fix the errors [edit]

Finally, e(x) is generated from i_k and e_{i_k} and then is subtracted from r(x) to get the sent message s(x).

Berlekamp–Massey decoder [edit]

The Berlekamp–Massey algorithm is an alternate iterative procedure for finding the error locator polynomial. During each iteration, it calculates a discrepancy based on a current instance of Λ(x) with an assumed number of errors *e*:

$$\Delta = S_i + \Lambda_1 \ S_{i-1} + \cdots + \Lambda_e \ S_{i-e}$$

and then adjusts Λ(x) and *e* so that a recalculated Δ would be zero. The article Berlekamp–Massey algorithm has a detailed description of the procedure. In the following example, C(x) is used to represent Λ(x).

Example [edit]

Consider the Reed–Solomon code defined in $GF(929)$ with $\alpha = 3$ and $t = 4$ (this is used in PDF417 barcodes). The generator polynomial is

$$g(x) = (x-3)(x-3^2)(x-3^3)(x-3^4) = x^4 + 809x^3 + 723x^2 + 568x + 522$$

If the message polynomial is $p(x) = 3\,x^2 + 2\,x + 1$, then the codeword is calculated as follows.

$$s_r(x) = p(x)\,x^t \mod g(x) = 547x^3 + 738x^2 + 442x + 455$$
$$s(x) = p(x)\,x^t - s_r(x) = 3x^6 + 2x^5 + 1x^4 + 382x^3 + 191x^2 + 487x + 474$$

Errors in transmission might cause this to be received instead.

$$r(x) = s(x) + e(x) = 3x^6 + 2x^5 + 123x^4 + 456x^3 + 191x^2 + 487x + 474$$

The syndromes are calculated by evaluating *r* at powers of *α*.

$$S_1 = r(3^1) = 3\cdot 3^6 + 2\cdot 3^5 + 123\cdot 3^4 + 456\cdot 3^3 + 191\cdot 3^2 + 487\cdot 3 + 474 = 732$$
$$S_2 = r(3^2) = 637,\ S_3 = r(3^3) = 762,\ S_4 = r(3^4) = 925$$

To correct the errors, first use the Berlekamp–Massey algorithm to calculate the error locator polynomial.

n	S_{n+1}	*d*	*C*	*B*	*b*	*m*
0	732	732	197 *x* + 1	1	732	1
1	637	846	173 *x* + 1	1	732	2
2	762	412	634 x^2 + 173 *x* + 1	173 *x* + 1	412	1
3	925	576	329 x^2 + 821 *x* + 1	173 *x* + 1	412	2

The final value of *C* is the error locator polynomial, Λ(*x*). The zeros can be found by trial substitution. They are x_1 = 757 = 3^{-3} and x_2 = 562 = 3^{-4}, corresponding to the error locations. To calculate the error values, apply the Forney algorithm.

$$\Omega(x) = S(x)\Lambda(x) \mod x^4 = 546x + 732$$
$$\Lambda'(x) = 658x + 821$$
$$e_1 = -\Omega(x_1)/\Lambda'(x_1) = -649/54 = 280 \times 843 = 74$$
$$e_2 = -\Omega(x_2)/\Lambda'(x_2) = 122$$

Subtracting e_1x^3 and e_2x^4 from the received polynomial *r* reproduces the original codeword *s*.

Euclidean decoder [edit]

Another iterative method for calculating the error locator polynomial is based on the Extended Euclidean algorithm .

Define S(x) , Λ(x) , and Ω(x) for *t* syndromes and *e* errors:

$$S(x) = S_t x^{t-1} + S_{t-1} x^{t-2} + \cdots + S_2 x + S_1$$
$$\Lambda(x) = \Lambda_e x^{e-1} + \Lambda_{e-1} x^{e-2} + \cdots + \Lambda_1 x + 1$$
$$\Omega(x) = \Omega_e x^{e-1} + \Omega_{e-1} x^{e-2} + \cdots + \Omega_1 x + \Omega_0$$

The key equation is:

$$\Lambda(x) S(x) = Q(x) x^t + \Omega(x)$$

For *t* = 6 and *e* = 3:

$$\begin{bmatrix} \Lambda_3 S_6 & x^8 \\ \Lambda_2 S_6 + \Lambda_3 S_5 & x^7 \\ \Lambda_1 S_6 + \Lambda_2 S_5 + \Lambda_3 S_4 & x^6 \\ S_6 + \Lambda_1 S_5 + \Lambda_2 S_4 + \Lambda_3 S_3 & x^5 \\ S_5 + \Lambda_1 S_4 + \Lambda_2 S_3 + \Lambda_3 S_2 & x^4 \\ S_4 + \Lambda_1 S_3 + \Lambda_2 S_2 + \Lambda_3 S_1 & x^3 \\ S_3 + \Lambda_1 S_2 + \Lambda_2 S_1 & x^2 \\ S_2 + \Lambda_1 S_1 & x \\ S_1 \end{bmatrix} = \begin{bmatrix} Q_2 x^8 \\ Q_1 x^7 \\ Q_0 x^6 \\ 0 \\ 0 \\ 0 \\ \Omega_2 x^2 \\ \Omega_1 x \\ \Omega_0 \end{bmatrix}$$

The middle terms are zero due to the relationship between Λ and syndromes.

The extended Euclidean algorithm can find a series of polynomials of the form

$A_i(x)\ S(x) + B_i(x)\ x^t = R_i(x)$

where the degree of R decreases as i increases. Once the degree of $R_i(x) < t/2$, then

$A_i(x) = \Lambda(x)$

$B_i(x) = -Q(x)$

$R_i(x) = \Omega(x)$.

B(x) and Q(x) don't need to be saved, so the algorithm becomes:

```
$R_{-1} = x^t$
$R_0 = S(x)$
$A_{-1} = 0$
$A_0 = 1$
i = 0
while degree of $R_i$ >= t/2
    i = i + 1
    $Q = R_{i-2} / R_{i-1}$
    $R_i = R_{i-2} - Q R_{i-1}$
    $A_i = A_{i-2} - Q A_{i-1}$
```

to set low order term of Λ(x) to 1, divide Λ(x) and Ω(x) by $A_i(0)$:

$\Lambda(x) = A_i / A_i(0)$
$\Omega(x) = R_i / A_i(0)$

$A_i(0)$ is the constant (low order) term of A_i.

Here is an example of the Euclidean method, using the same data as the Berlekamp Massey example above.

i	R_i	A_i
-1	$001\ x^4 + 000\ x^3 + 000\ x^2 + 000\ x + 000$	000
0	$925\ x^3 + 762\ x^2 + 637\ x + 732$	001
1	$683\ x^2 + 676\ x + 024$	$697\ x + 396$
2	$673\ x + 596$	$608\ x^2 + 704\ x + 544$

$\Lambda(x) = A_2 / 544 = 329\ x^2 + 821\ x + 001$
$\Omega(x) = R_2 / 544 = 546\ x + 732$

Decoding in frequency domain (sketch) [edit]

The above algorithms are presented in the time domain. Decoding in the frequency domain, using Fourier transform techniques, can offer computational and implementation advantages. (Hong & Vetterli 1995)

The following is a sketch of the main idea behind this error correction technique.

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.

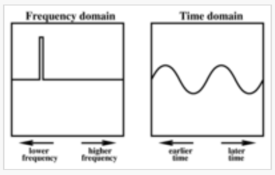

Figure 1

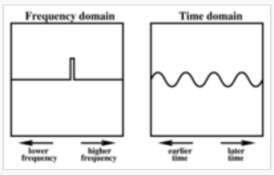

Figure 2

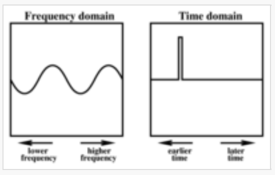

Figure 3

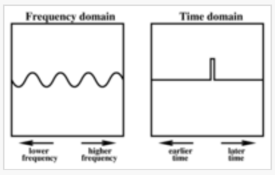

Figure 4

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.

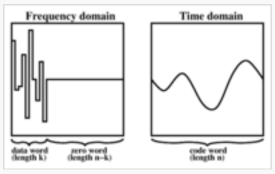

Figure 5

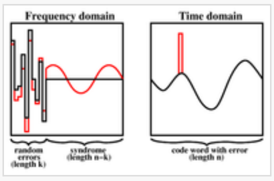

Figure 6

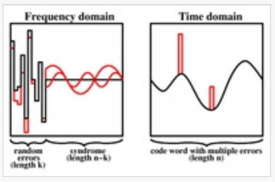

Figure 7

Decoding beyond the error-correction bound [edit]

The Singleton bound states that the minimum distance *d* of a linear block code of size (*n*,*k*) is upper-bounded by *n* − *k* + 1. The distance *d* was usually understood to limit the error-correction capability to ⌊*d*/2⌋. The Reed–Solomon code achieves this bound with equality, and can thus correct up to ⌊(*n* − *k* + 1)/2⌋ errors. However, this error-correction bound is not exact.

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.[7] It applies to Reed–Solomon codes and more generally to algebraic geometric 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.

Soft-decoding [edit]

The algebraic decoding methods described above are hard-decision methods, which means that for every symbol a hard decision is made about its value. For example, a decoder could associate with each symbol an additional value corresponding to the channel demodulator's confidence in the correctness of the symbol. The advent of LDPC and turbo codes, which employ iterated soft-decision belief propagation decoding methods to achieve error-correction performance close to the theoretical limit, has spurred interest in applying soft-decision decoding to conventional algebraic codes. In 2003, Ralf Koetter and Alexander Vardy presented a polynomial-time soft-decision algebraic list-decoding algorithm for Reed–Solomon codes, which was based upon the work by Sudan and Guruswami.[8]

Matlab Example [edit]

Encoder [edit]

Here we present a simple Matlab implementation for an encoder.

```
function [ encoded ] = rsEncoder( msg, m, prim_poly,n,k )
%RSENCODER Encode message with the Reed-Solomon algorithm
% m is the number of bits per symbol
% prim_poly: Primitive polynomial p(x). Ie for DM is 301
% k is the size of the message
% n is the total size (k+redundant)
% Example: msg = uint8('Test')
% enc_msg = rsEncoder(msg,8,301,12,numel(msg));

% Get the alpha
alpha = gf(2,m,prim_poly);

% Get the reed-solomon generating polynomial g(x)
g_x = genpoly(k,n,alpha);

% Multiply the information by X^(n-k), or just pad with zeros at the end to
% get space to add the redundant information
msg_padded = gf([msg zeros(1,n-k)],m,prim_poly);

% Get the reminder of the division of the extended message by the
% reed-solomon generating polynomial g(x)
[~,reminder] = deconv(msg_padded,g_x);

% Now return the message with the redundant information
encoded = msg_padded - reminder;

end

% Find the reed-solomon generating polynomial g(x), by the way this is the
% same as the rsgenpoly function on matlab
function g = genpoly(k, n, alpha)
g = 1;
% A multiplication on the galois field is just a convolution
for k = mod(1:n-k,n)
    g = conv(g, [1 alpha .^ (k)]);
end
end
```

Decoder [edit]

Now the decoding part:

```
function [ decoded, error_pos, error_mag, g, S ] = rsDecoder( encoded, m,prim_poly,n,k )
%RSDECODER Decode a reed-solomon encoded message
%   Example:
% [dec,~,~,~,~] = rsDecoder(enc_msg,8,301,12,numel(msg))
max_errors = floor((n-k)/2);
orig_vals = encoded.x;
% Initialize the error vector
errors = zeros(1,n);
g = [];
S = [];

% Get the alpha
alpha = gf(2,m,prim_poly);

% Find the syndromes (Check if dividing the message by the generator
% polynomial the result is zero)
Synd = polyval(encoded, alpha .^(1:n-k));
Syndromes = trim(Synd);

% If all syndromes are zeros (perfectly divisible) there are no errors
if isempty(Syndromes.x)
    decoded = orig_vals(1:k);
    error_pos = [];
    error_mag = [];
    g = [];
    S = Synd;
    return;
end

% Prepare for the euclidean algorithm (Used to find the error locating
% polynomials)
r0 = [1, zeros(1,2*max_errors)]; r0 = gf(r0,m,prim_poly); r0 = trim(r0);
size_r0 = length(r0);
r1 = Syndromes;
f0 = gf([zeros(1,size_r0-1) 1],m,prim_poly);
f1 = gf(zeros(1,size_r0),m,prim_poly);
```

```
g0 = f1; g1 = f0;

% Do the euclidian algorithm on the polynomials r0(x) and Syndromes(x) in
% order to find the error locating polynomial
while true
    % Do a long division
    [quocient,remainder] = deconv(r0,r1);
    % Add some zeros
    quocient = pad(quocient,length(g1));

    % Find quocient*g1 and pad
    c = conv(quocient,g1);
    c = trim(c);
    c = pad(c,length(g0));

    % Update g as g0-quocient*g1
    g = g0 - c;

    % Check if the degree of remainder(x) is less than max_errors
    if all(remainder(1:end - max_errors)==0)
        break;
    end

    % Update r0,r1,g0,g1 and remove leading zeros
    r0 = trim(r1); r1 = trim(remainder);
    g0 = g1; g1 = g;
end

% Remove leading zeros
g = trim(g);

% Find the zeros of the error polynomial on this galois field
evalPoly = polyval(g,alpha .^(n-1:-1:0));
error_pos = gf(find(evalPoly == 0),m);

% If no error position is found we return the received work, because
% basically is nothing that we could do and we return the received message
if isempty(error_pos)
    decoded = orig_vals(1:k);
    error_mag = [];
    return;
end

% Prepare a linear system to solve the error polynomial and find the error
% magnitudes
size_error = length(error_pos);
Syndrome_Vals = Syndromes.x;
b(:,1) = Syndrome_Vals(1:size_error);
for idx=1:size_error
    e = alpha .^ (idx*(n-error_pos.x));
    err = e.x;
    er(idx,:) = err;
end

% Solve the linear system
error_mag = (gf(er,m,prim_poly) \ gf(b,m,prim_poly))';
% Put the error magnitude on the error vector
errors(error_pos.x) = error_mag.x;
% Bring this vector to the galois field
errors_gf = gf(errors,m,prim_poly);

% Now to fix the errors just add with the encoced code
decoded_gf = encoded(1:k) + errors_gf(1:k);
decoded = decoded_gf.x;

end

% Remove leading zeros from galois array
function gt = trim(g)
gx = g.x;
gt = gf(gx(find(gx,1):end),g.m,g.prim_poly);
end

% Add leading zeros
function xpad = pad(x,k)
len = length(x);
if (len<k)
    xpad = [zeros(1,k-len) x];
end
end
```

See also [edit]

- BCH code
- Cyclic code
- Chien search

- Berlekamp–Massey algorithm
- Forward error correction
- Berlekamp–Welch algorithm
- Folded Reed–Solomon code

Notes [edit]

1. ^ Reed & Solomon (1960)
2. ^ Immink, K. A. S. (1994), "Reed–Solomon Codes and the Compact Disc", in Wicker, Stephen B.; Bhargava, Vijay K., *Reed–Solomon Codes and Their Applications*, IEEE Press, ISBN 978-0-7803-1025-4
3. ^ Not quite true. See remarks below.
4. ^ Lidl, Rudolf; Pilz, Günter (1999). *Applied Abstract Algebra* (2nd ed.). Wiley. p. 226.
5. ^ See Lin & Costello (1983, p. 171), for example.
6. ^ Pfender, Florian; Ziegler, Günter M. (September 2004), "Kissing Numbers, Sphere Packings, and Some Unexpected Proofs" (PDF), *Notices of the American Mathematical Society* **51** (8): 873–883. Explains the Delsarte-Goethals-Seidel theorem as used in the context of the error correcting code for compact disc.
7. ^ Guruswami, V.; Sudan, M. (September 1999), "Improved decoding of Reed–Solomon codes and algebraic geometry codes", *IEEE Transactions on Information Theory* **45** (6): 1757–1767, doi:10.1109/18.782097
8. ^ Koetter, Ralf; Vardy, Alexander (2003). "Algebraic soft-decision decoding of Reed–Solomon codes". *IEEE Transactions on Information Theory* **49** (11): 2809–2825. doi:10.1109/TIT.2003.819332.

References [edit]

- Cipra, Barry A. (1993), "The Ubiquitous Reed–Solomon Codes", *SIAM News* **26** (1)
- Berlekamp, Elwyn R. (1967), *Nonbinary BCH decoding*, International Symposium on Information Theory, San Remo, Italy
- Berlekamp, Elwyn R. (1984) [1968], *Algebraic Coding Theory* (Revised ed.), Laguna Hills, CA: Aegean Park Press, ISBN 0-89412-063-8
- Forney, Jr., G. (October 1965), "On Decoding BCH Codes", *IEEE Transactions on Information Theory* **11** (4): 549–557, doi:10.1109/TIT.1965.1053825
- Gill, John, *EE387 Notes #7, Handout #28* (PDF), Stanford University, retrieved April 21, 2010
- Hong, Jonathan; Vetterli, Martin (August 1995), "Simple Algorithms for BCH Decoding", *IEEE Transactions on Communications* **43** (8): 2324–2333, doi:10.1109/26.403765
- Koetter, Ralf (2005), *Reed–Solomon Codes*, MIT Lecture Notes 6.451 (Video)
- Lin, Shu; Costello, Jr., Daniel J. (1983), *Error Control Coding: Fundamentals and Applications*, New Jersey, NJ: Prentice-Hall, ISBN 0-13-283796-X
- MacWilliams, F. J.; Sloane, N. J. A. (1977), *The Theory of Error-Correcting Codes*, New York, NY: North-Holland Publishing Company
- Massey, J. L. (1969), "Shift-register synthesis and BCH decoding" (PDF), *IEEE Transactions on Information Theory*, IT-15 (1): 122–127
- Peterson, Wesley W. (1960), "Encoding and Error Correction Procedures for the Bose-Chaudhuri Codes", *IRE Transactions on Information Theory* (Institute of Radio Engineers) **IT–6**: 459–470
- Reed, Irving S.; Chen, Xuemin (1999), *Error-Control Coding for Data Networks*, Boston, MA: Kluwer Academic Publishers
- Reed, Irving S.; Solomon, Gustave (1960), "Polynomial Codes over Certain Finite Fields", *Journal of the Society for Industrial and Applied Mathematics (SIAM)* **8** (2): 300–304, doi:10.1137/0108018
- Welch, L. R. (1997), *The Original View of Reed–Solomon Codes* (PDF), Lecture Notes

External links [edit]

Information and Tutorials [edit]

- Introduction to Reed-Solomon codes: principles, architecture and implementation (CMU)
- A Tutorial on Reed–Solomon Coding for Fault-Tolerance in RAID-like Systems
- Algebraic soft-decoding of Reed–Solomon codes
- Wikiversity: Reed–Solomon codes for coders
- BBC R&D White Paper WHP031
- Geisel, William A. (August 1990), *Tutorial on Reed–Solomon Error Correction Coding* (PDF), Technical Memorandum, NASA, TM-102162

Code [edit]

- Schifra Open Source C++ Reed–Solomon Codec
- Henry Minsky's RSCode library, Reed–Solomon encoder/decoder
- Open Source C++ Reed-Solomon Soft Decoding library
- Matlab implementation of errors and-erasures Reed–Solomon decoding
- Pure-Python implementation of a Reed–Solomon codec

Categories: Error detection and correction | Coding theory