Jerasure: A Library in C/C++ Facilitating Erasure Coding for Storage Applications

Version 2.0

James S. Plank* Scott Simmerman Catherine D. Schuman

Technical Report CS-08-627
Department of Electrical Engineering and Computer Science
University of Tennessee
Knoxville, TN 37996

`http://www.cs.utk.edu/~plank/plank/papers/CS-08-627.html`

This describes revision 2.0 of the code.

Abstract

This paper descibes version 2.0 of **jerasure**, a library in C++ that supports erasure coding in storage applications. In this paper, we describe both the techniques and algorithms, plus the interface to the code. Thus, this serves as a quasi-tutorial and a programmer's guide.

Version 2.0 of jerasure is written in C++, uses a new object-oriented interface, adds generalized EVENODD and generalized RDP to the library, supports multi-threaded coding, and includes two new example applications.

If You Use This Library or Document

Please send me an email to let me know how it goes. One of the ways in which I am evaluated both internally and externally is by the impact of my work, and if you have found this library and/or this document useful, I would like to be able to document it. Please send mail to *plank@eecs.utk.edu*.

The library itself is protected by the GNU LGPL. It is free to use and modify within the bounds of the LGPL. None of the techniques implemented in this library have been patented.

Finding the Code

Please see `http://www.cs.utk.edu/~plank/plank/papers/CS-08-627.html` to get the TAR file for this code.

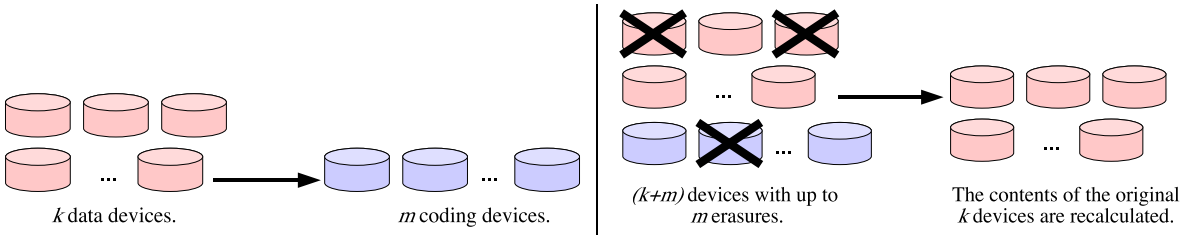
*`plank@cs.utk.edu` or `plank@eecs.utk.edu`, 865-974-4397, This material is based upon work supported by the National Science Foundation under grant CNS-0615221.

Contents

1 Introduction

Erasure coding for storage applications is growing in importance as storage systems grow in size and complexity. This paper describes **jerasure**, a library in C++ that supports erasure coding applications. **Jerasure** has been designed to be modular, fast and flexible. It is our hope that storage designers and programmers will find **jerasure** to be a convenient tool to add fault tolerance to their storage systems.

Jerasure supports a *horizontal* mode of erasure codes. We assume that we have k devices that hold data. To that, we will add m devices whose contents will be calculated from the original k devices. If the erasure code is a *Maximum Distance Separable (MDS)* code, then the entire system will be able to tolerate the loss of any m devices.



(a) Encoding. (b) Decoding.

Figure 1: The act of *encoding* takes the contents of k data devices and encodes them on m coding devices. The act of *decoding* takes some subset of the collection of $(k + m)$ total devices and from them recalcalates the original k devices of data.

As depicted in Figure 1, the act of encoding takes the original k data devices, and from them calculates m coding devices. The act of decoding takes the collection of $(k + m)$ devices with erasures, and from the surviving devices recalculates the contents of the original k data devices.

Most codes have a third parameter w, which is the *word size*. The description of a code views each device as having w bits worth of data. The data devices are denoted D_0 through D_{k-1} and the coding devices are denoted C_0 through C_{m-1}. Each device D_i or C_j holds w bits, denoted $d_{i,0}, \ldots d_{i,w-1}$ and $c_{i,0}, \ldots c_{i,w-1}$. In reality of course, devices hold megabytes of data. To map the description of a code to its realization in a real system, we do one of two things:

1. When $w \in \{8, 16, 32\}$, we can consider each collection of w bits to be a byte, short word or word respectively. Consider the case when $w = 8$. We may view each device to hold B bytes. The first byte of each coding device will be encoded with the first byte of each data device. The second byte of each coding device will be encoded with the second byte of each data device. And so on. This is how Standard Reed-Solomon coding works, and it should be clear how it works when $w = 16$ or $w = 32$.

2. Most other codes work by defining each coding bit $c_{i,j}$ to be the bitwise exclusive-or (XOR) of some subset of the other bits. To implement these codes in a real system, we assume that the device is composed of w *packets* of equal size. Now each packet is calculated to be the bitwise exclusive-or of some subset of the other packets. In this way, we can take advantage of the fact that we can perform XOR operations on whole computer words rather than on bits.

   The process is illustrated in Figure 2. In this figure, we assume that $k = 4$, $m = 2$ and $w = 4$. Suppose

that a code is defined such that coding bit $c_{1,0}$ is goverened by the equation:

$$c_{1,0} = d_{0,0} \oplus d_{1,1} \oplus d_{2,2} \oplus d_{3,3},$$

where $\oplus$ is the XOR operation. Figure 2 shows how the coding packet corresponding to $c_{1,0}$ is calculated from the data packets corresponding to $d_{0,0}$, $d_{1,1}$, $d_{2,2}$ and $d_{3,3}$. We call the size of each packet the *packet size*, and the size of w packets to be the *coding block size*. The packetsize must be a multiple of 8 so obviously, the coding block size will be a multiple of $w * packetsize$.

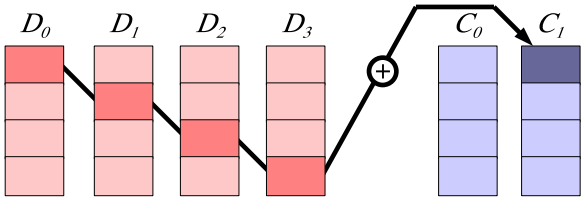


Figure 2: Although codes are described on systems of w bits, their implementation employs *packets* that are much larger. Each packet in the implementation corresponds to a bit of the description. This figure is showing how the equation $c_{1,0} = d_{0,0} \oplus d_{1,1} \oplus d_{2,2} \oplus d_{3,3}$ is realized in an implementation.

2 The Modules of the Library

This library is broken into four modules, each with its own header file and implementation in C++. Typically, when using a code, one only needs three of these modules: **galois**, **jerasure** and one of the others. The modules are:

1. **galois.h/galois.cpp**: These are procedures for Galois Field Arithmetic as described and implemented in [Pla07].

2. **jerasure.h/jer_slices.cpp, jer_gen_t.cpp, jer_matrix.cpp**: These are kernel routines that are common to most erasure codes. They do not depend on any module other than **galois**. They include support for matrix-based coding and decoding, bit-matrix-based coding and decoding, conversion of bit-matrices to schedules, matrix and bit-matrix inversion.

3. **reedsol.h/reedsol.cpp**: These are procedures for creating distribution matrices for Reed-Solomon coding [RS60, Pla97, PD05], including Cauchy Reed-Solomon coding [BKK+95, PX06]. They also include the optimized version of Reed-Solomon encoding for RAID-6 as discussed in [Anv07].

4. **bitmatrices.h/bitmatrices.cpp**: These are procedures for performing encoding and decoding with generalized EVENODD [BBV], generalized RDP, and minimal density MDS codes – the RAID-6 Liberation codes [Pla08b], Blaum-Roth codes [BR99] and the RAID-6 Liber8tion code [Pla08a].

Each module is described in its own section below. Additionally, there are example programs that show the usage of each module.

3 Matrix-Based Coding In General

The mechanics of matrix-based coding are explained in great detail in [Pla97]. We give a high-level overview here.

> **Authors' Caveat:** *We are using old nomenclature of "distribution matrices." In standard coding theory, the "distribution matrix" is the transpose of the Generator matrix. In the next revision of* **jerasure***, we will update the nomenclature to be more consistent with classic coding theory.*

Suppose we have k data words and m coding words, each composed of w bits. We can describe the state of a matrix-based coding system by a matrix-vector product as depicted in Figure 3. The matrix is called a *distribution matrix* and is a $(k+m) \times k$ matrix. The elements of the matrix are numbers in $GF(2^w)$ for some value of w. This means that they are integers between 0 and $2^w - 1$, and arithmetic is performed using Galois Field arithmetic: addition is equal to XOR, and multiplication is implemented in a variety of ways. The Galois Field arithmetic library in [Pla07] has procedures which implement Galois Field arithmetic.

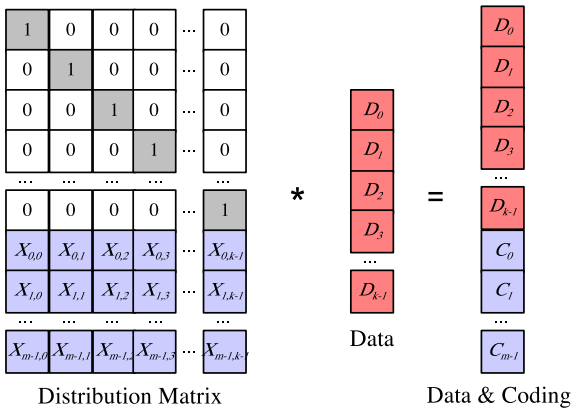


Figure 3: Using a matrix-vector product to describe a coding system.

The top k rows of the distribution matrix compsose a $k \times k$ identity matrix. The remaining m rows are called the *coding matrix*, and are defined in a variety of ways [Rab89, Pre89, BKK+95, PD05]. The distribution matrix is multiplied by a vector that contains the data words and yields a product vector containing both the data and the coding words. Therefore, to encode, we need to perform m dot products of the coding matrix with the data.

To decode, we note that each word in the system has a corresponding row of the distribution matrix. When devices fail, we create a decoding matrix from k rows of the distribution that correspond to non-failed devices. Note that this matrix multiplied by the original data equals the k survivors whose rows we selected. If we invert this matrix and multiply it by both sides of the equation, then we are given a decoding equation – the inverted matrix multiplied by the survivors equals the original data.

4 Bit-Matrix Coding In General

Bit-matrix coding is first described in the original Cauchy Reed-Solomon coding paper [BKK+95]. To encode and decode with a bit-matrix, we expand a distribution matrix in $GF(2^w)$ by a factor of w in each direction to yield a $w(k+m) \times wk$ matrix which we call a *binary distribution matrix (BDM)*. We multiply that by a wk element vector, which is composed of w bits from each data device. The product is a $w(k+m)$ element vector composed of w bits from each data and coding device. This is depicted in Figure 4. It is useful to visualize the matrix as being composed of $w \times w$ sub-matrices.

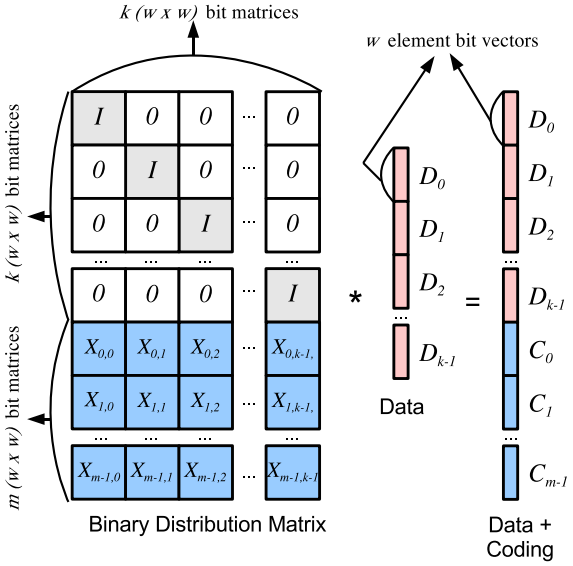


Figure 4: Describing a coding system with a bit-matrix-vector product.

As with the matrix-vector product in $GF(2^w)$, each row of the product corresponds to a row of the BDM, and is computed as the dot product of that row and the data bits. Since all elements are bits, we may perform the dot product by taking the XOR of each data bit whose element in the matrix's row is one. In other words, rather than performing the dot product with additions and multiplications, we perform it only with XORs. Moreover, the performance of this dot product is directly related to the number of ones in the row. Therefore, it behooves us to find matrices with few ones.

Decoding with bit-matrices is the same as with matrices over $GF(2^w)$, except now each device corresponds to w rows of the matrix, rather than one. Also keep in mind that a bit in this description corresponds to a packet in the implementation.

While the classic construction of bit-matrices starts with a standard distribution matrix in $GF(2^w)$, it is possible to construct bit-matrices that have no relation to Galois Field arithmetic yet still have desired coding and decoding properties. The minimal density RAID-6 codes work in this fashion.

4.1 Using a schedule rather than a bit-matrix

Consider the act of encoding with a bit-matrix. We give an example in Figure 5, where $k = 3$, $w = 5$, and we are calculating the contents of one coding device. The straightforward way to encode is to calculate the five dot products for each of the five bits of the coding device, and we can do that by traversing each of the five rows, performing XORs where there are ones in the matrix.

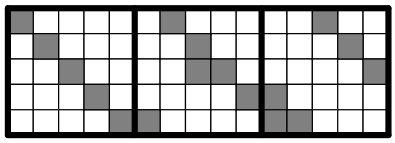


Figure 5: An example super-row of a bit-matrix for $k = 3$, $w = 5$.

Since the matrix is sparse, it is more efficient to precompute the coding operations, rather than traversing the matrix each time one encodes. The data structure that we use to represent encoding is a *schedule*, which is a list of 5-tuples:

$$< op, s_d, s_b, d_d, d_b >,$$

where *op* is an operation code: 0 for copy and 1 for XOR, s_d is the id of the source device and s_b is the bit of the source device. The last two elements, d_d and d_b are the destination device and bit. By convention, we identify devices using integers from zero to $k + m - 1$. An id $i < k$ identifies data device D_i, and an id $i \geq k$ identifies coding device C_{i-k}.

A schedule for encoding using the bit-matrix in Figure 5 is shown in Figure 6.

(a)	(b)
$< 0, 0, 0, 3, 0 >, < 1, 1, 1, 3, 0 >, < 1, 2, 2, 3, 0 >,$	$c_{0,0} = d_{0,0} \oplus d_{1,1} \oplus d_{2,2}$
$< 0, 0, 1, 3, 1 >, < 1, 1, 2, 3, 1 >, < 1, 2, 3, 3, 1 >,$	$c_{0,1} = d_{0,1} \oplus d_{1,2} \oplus d_{2,3}$
$< 0, 0, 2, 3, 2 >, < 1, 1, 2, 3, 2 >, < 1, 1, 3, 3, 2 >, < 1, 2, 4, 3, 2 >,$	$c_{0,2} = d_{0,2} \oplus d_{1,2} \oplus d_{1,3} \oplus d_{2,4}$
$< 0, 0, 3, 3, 3 >, < 1, 1, 4, 3, 3 >, < 1, 2, 0, 3, 3 >,$	$c_{0,3} = d_{0,3} \oplus d_{1,4} \oplus d_{2,0}$
$< 0, 0, 4, 3, 4 >, < 1, 1, 0, 3, 4 >, < 1, 2, 0, 3, 4 >, < 1, 2, 1, 3, 4 > .$	$c_{0,4} = d_{0,4} \oplus d_{1,0} \oplus d_{2,0} \oplus d_{2,1}$

Figure 6: A schedule of bit-matrix operations for the bit-matrix in Figure 5. (a) shows the schedule, and (b) shows the dot-product equations corresponding to each line of the schedule.

As noted in [HDRT05, Pla08b], one can derive schedules for bit-matrix encoding and decoding that make use of common expressions in the dot products, and therefore can perform the bit-matrix-vector product with fewer XOR operations than simply traversing the bit-matrix. This is how RDP encoding works with optimal performance [CEG+04], even though there are more than kw ones in the last w rows of its BDM. We term such scheduling *smart* scheduling, and scheduling by simply traversing the matrix *dumb* scheduling.

5 MDS Codes

A code is MDS if it can recover the data following the failure of any m devices. If a matrix-vector product is used to define the code, then it is MDS if every combination of k rows composes an invertible matrix. If a bit-

matrix is used, then we define a *super-row* to be a row's worth of $w \times w$ submatrices. The code is MDS if every combination of k super-rows composes an invertible matrix. Again, one may generate an MDS code using standard techniques such as employing a Vandermonde matrix [PD05] or Cauchy matrix [Rab89, BKK+95]. However, there are other constructions that also yield MDS matrices, such as EVENODD coding [BBBM95], RDP coding [CEG+04], the STAR code [HX05], Feng's codes [FDBS05a, FDBS05b] and the minimal density RAID-6 codes [BR99, Pla08a, Pla08b].

6 Part 1 of the Library: Galois Field Arithmetic (galois.h)

The files **galois.h** and **galois.cpp** contain procedures for Galois Field arithmetic in $GF(2^w)$ for $1 \leq w \leq 32$. There are functions for performing single arithmetic operations, XOR-ing a region of bytes, and performing multiplication of a region of bytes by a constant in $GF(2^8)$, $GF(2^{16})$ and $GF(2^{32})$. All of these procedures are defined in a separate technical report which focuses solely on Galois Field arithmetic [Pla07]. The following section lists the Galois functions used by **jerasure**.

6.1 Galois procedures used in Jerasure - galois.cpp

- **galois_single_multiply(int a, int b, int w)** and **galois_single_divide(int a, int b, int w)**: These perform multiplication and division on single elements **a** and **b** of $GF(2^{\mathbf{w}})$.
- **galois_region_xor(char *r1, char *r2, char *r3, int nbytes)**: This XORs two regions of bytes, **r1** and **r2**, and places the sum in **r3**. Note that **r3** may be equal to **r1** or **r2** if we are replacing one of the regions by the sum. **Nbytes** must be a multiple of **8**.
- **galois_w08_region_multiply(unsigned char *region, int multby, int nbytes, unsigned char *r2, int add)**: This multiplies an entire region of bytes by the constant **multby** in $GF(2^8)$. **Region** is overwritten if **r2** is **NULL**. Otherwise, if **add** is zero, the products are placed in **r2**. If **add** is non-zero, then the products are XOR'd with the bytes in **r2**. **Nbytes** must be a multiple of **8**.
- **galois_w16_region_multiply()** and **galois_w32_region_multiply()** are identical to **galois_w08_region_multiply()**, except they are in $GF(2^{16})$ and $GF(2^{32})$ respectively. **Nbytes** must still be a multiple of **8**.
- **galois_w08_region_multby_2(char *region, int nbytes)**: This performs the fast multiplication by two in $GF(2^8)$ using Anvin's optimization [Anv07]. **Nbytes** must be a multiple of **8**.
- **galois_w16_region_multby_2()** and **galois_w32_region_multby_2()** are identical to **galois_w08_region_multby_2()**, except they are in $GF(2^{16})$ and $GF(2^{32})$ respectively. **Nbytes** must still be a multiple of **8**.

6.2 Example programs

- **galois_01.cpp**: This simply demonstrates doing fast multiplication by two in $GF(2^w)$ for $w \in \{8, 16, 32\}$. It has one parameter: w.

```
UNIX> galois_01 16
Short  0: 58899 *2 = 56365
Short  1: 54100 *2 = 46755
```

```
Short  2: 64788 *2 = 59939
Short  3: 52269 *2 = 34897
Short  4: 47389 *2 = 25137
Short  5: 42657 *2 = 23881
Short  6: 49248 *2 = 37067
Short  7:  4812 *2 =  9624
UNIX>
```

This demonstrates usage of **galois_w08_region_multby_2()**, **galois_w16_region_multby_2()** and **galois_w32_region_multby_2()**.

7 Part 2 of the Library: Kernel Classes (jerasure.h)

Jerasure.h contains prototypes of the functions necessary for coding. This header file also specifies the different classes used in the encoding process. Function declarations are split into object files based on the class with which they are associated. Each class is explained individually in the following section. We will decribe the procedures that compose **jerasure.h** by the .cpp files in which they are found. In Section 7.5, we demonstrate the usage of these classes with example programs.

7.1 The JER_Region class - jerasure.h

The most basic object defined in jerasure.h is the JER_Region class. This class contains three properties which describe a region of a disk. JER_Region objects are used by various methods of the JER_Slices class. The class properties are listed below.

- **int drive**: The id of the drive containing this region (numbered 0 to **N**-1).
- **int start**: The index of the first byte in this region.
- **int size**: The number of bytes in this region.

7.2 Encoding and decoding data - jer_slices.cpp

The JER_Slices class links the user data to a generator and manages the drive states. JER_Slices also contains the methods related to encoding and decoding data. The class contains the following variables:

- **int K**: The number of data devices.
- **int N**: The total number of devices. This is the sum of the number of data devices (**K**) and the number of coding devices (**M**). The number of coding devices is not stored, because it is calculated as **N**-**K**.
- **JER_Gen_T *G**: The generator used to encode and decode the data.
- **int PacketSize**: The packet size as defined in section 1. This must be a multiple of **8**.
- **int PacketsPerSlice**: The number of packets in a single device. This must be a multiple of **G→WPD**.

- **vector ⟨unsigned char *⟩ Ptrs**: This is an array of pointers with one element per device. Each pointer should point to **PacketSize** * **PacketsPerSlice** bytes of data. **Ptrs** is initialized in the **JER_Slices** constructors. For constructors with a **ptrs** argument, **Ptrs** is set to a copy of **ptrs**. Otherwise, **Ptrs** is resized to size **N**.

- **int DataOnly**: This variable is only used for vertical codes. It is set by the user in one of the **JER_Slices** constructors. When **DataOnly** $\neq 0$, it is assumed that the user-provided **Ptrs** do not contain encoded data. Therefore, all **N** drives are set to the *down* state. If **DataOnly** $= 0$, it is assumed that **Ptrs** contains encoded data, and all **N** drives are set to the *up* state.

- **vector ⟨int⟩ States**: This vector describes whether a particular drive is *up*, *down*, or *unusable*. The state of the i^{th} drive is stored in **States**[i]. Valid values for elements of **States** include 0, 1, or 2, corresponding respectively to a drive being *up*, *down*, or *unusable*. The **JER_Slices** constructors resize **States** to size **N**.

- **int NumberOfCores**: The number of cores that the user's machine has. This variable is set to a default value of 1 in the **JER_Slices** constructors. However, users may update the value at any time. Some procedures, such as encoding or decoding with a bitmatrix, will attempt to create a thread for each core. Each thread will work independently on its task in order to increase performance.

- **string MultiThreadMethod**: This variable holds the name of the technique to be used when performing multi-threaded dot products on bit-matrices. The technique names describe the amount of data that each thread will work with. Valid values include *disks*, *packet_rows*, *packet_cols*, or *packets*. An undefined value will default to the technique that handles the largest chunks of data, while allowing a separate thread for each core. For example, if we are encoding one drive with **NumberOfCores** = 4 and **MultiThreadMethod** = "", **MultiThreadMethod** will not default to *disks*. This is because only one core would be utilized. The actual method chosen in this situation depends on **PacketSize** and **PacketsPerSlice**.

- **int XORs**: The total number of bytes that have been xorred. This is incremented after methods call **galois_region_xor()**.

- **int GF_Mults**: The total number of bytes that have been multiplied by a constant in $GF(2^w)$. This is incremented when methods call **galois_w08_region multiply()**, **galois_w16_region_multiply()**, **galois_w32_region_multiply()**, **galois_w08_region_multby_2()**, **galois_w16_region_multby_2()**, or **galois_w32_region_multby_2()**.

- **int Memcpys**: The number of bytes that have been copied using memcpy().

The member functions of JER_Slices handle the manipulation of the data. **All constructors initialize XORs, GF_Mults, and Memcpys to 0, NumberOfCores to 1, and MultiThreadMethod to "".** The following functions are members of the class.

- **JER_Slices(int n, int k, int ps, int pps, JER_Gen_T *g, vector ⟨unsigned char *⟩ ptrs, int data_only)**: This constructor is used when data is setup in a format suitable for vertical coding. The **ptrs** argument should contain **N** pointers to memory allocated for the data. If **data_only** $= 0$, it is assumed that the data pointed to by **ptrs** is already properly encoded. All drives are given a state of *up*. If **data_only** is non-zero, it is assumed that the data is not encoded. All drives are given a state of *down*. **Encode()** is not called in this constructor.

- **JER_Slices(int n, int k, int ps, int pps, JER_Gen_T *g, vector ⟨unsigned char *⟩ ptrs)**: This constructor copies the **ptrs** argument to **Ptrs**. The first **K** drive states are *up*, and the coding drives are set to *down*. **Encode()** is then called, and the coding drives are set to *up* if encoding occurs successfully.

- **JER_Slices(int n, int k, int ps, int pps, JER_Gen_T *g)**: This constructor resizes **Ptrs** to contain **N** elements. The first **K** drives are set to *up*, and the coding drives are set to *down*. **Encode()** is not called.

- **void Add_Partial_Failure(int drive, int start, int size)**: Creates a new **JER_Region** object, sets its values, and appends it to **Pfs**.

- **void Add_Partial_Failure(JER_Region &r)**: Appends the **JER_Region** argument to **Pfs**.

- **void Add_Drive_Failure(int drive)**: Assigns a state of *down* to the device with an id of **drive** (**States**[**drive**] = 1).

- **int Recover_Partial_Failures()**: For each partial failure in **Pfs**, this calls Dotprod and attempts to recover from the failure. If the partial failure is fixed, the **JER_Region** element is remove from **Pfs**. This method returns the number of remaining JER_Regions in **Pfs**.

- **int Update_Region(int drive, int start, int size, unsigned char *new_data)**: This method updates the region of the drive with an id of **drive**. If a data drive is updated, it attempts to re-encode the corresponding regions of all **M** coding drives by calling **Recover_Partial_Failures()**. If a coding drive is updated, the function will always return 0. If a data drive is updated, this function returns the number of partial failures remaining in the drives. Therefore, the function returns 0 on success.

- **int Update_Region(JER_Region &r, unsigned char *new_data)**: This method updates the region of the drive with an id of **drive**. If a data drive is updated, it attempts to re-encode the corresponding regions of all **M** coding drives by calling **Recover_Partial_Failures()**. If a coding drive is updated, the function will always return 0. If a data drive is updated, this function returns the number of partial failures remaining in the drives. Therefore, the function returns 0 on success.

- **int Encode()**: **Encode** first calls **Recover_Partial_Failures()**. **Encode** returns -1 if some partial failures could not be recovered from.

  If the generator matrix is a bitmatrix (**G**→**M**→**W** = 1), the procedure first tries to find an encoding schedule. If the schedule is found, it is used to encode. Otherwise, encoding is performed by calling **Dotprod()**.For generator matrices with **W** $\neq$ 1, this encodes with a matrix in $GF(2^w)$ as described in Section 3 above. If (**N**-**K**) = 2, and the generator's **rs_r6** property is *true*, this encodes using Anvin's optimization [Anv07]. Encoding via Anvin's method does not require the generator to have a matrix, because the coding matrix is implicit.

  Sets all coding drive states to *up* and returns 0 on success; sets all drive states to *down* and returns -1 on failure.

- **int Decode()**: **Decode** first calls **Recover_Partial_Failures()**. **Decode** returns -1 if some partial failures could not be recovered from.

If the generator matrix is a bitmatrix (**G→M→W** = 1) and (**N-K**) = 2, the procedure tries to find the schedule matching the current drive states. If the schedule is found, it is used to decode. Otherwise, decoding is performed by calling **Dotprod()**. For generator matrices with **W** $\neq$ 1, this decodes using a matrix in $GF(2^w)$, **W** $\in \{8, 16, 32\}$. This works by creating a decoding matrix and performing the matrix/vector product, then re-encoding any erased coding devices. When it is done, the decoding matrix is discarded. If you want access to the decoding matrix, you should use the **Make_Decoding_Matrix()** method listed below.

Sets all drive states to *up* and returns 0 on success; leaves drive states unaltered and returns -1 on failure.

- **int Decode_Schedule_Lazy(int smart)**: This decodes the data using a schedule. It first searches **G→Schedules** for a schedule matching the current drive states. If a suitable schedule is not found, it is created. If **smart** = 1, the newly created schedule's type is CSHR. The new schedule is deleted once decoding is complete, and it is not inserted into **G→Schedules**. The function will fail if the generator matrix is not a bitmatrix (if **G→W** $\neq$ 1). Sets all drive states to *up* and returns 0 on success; leaves drive states unaltered and returns -1 on failure.
- **int Dotprod(JER_Matrix *jm, vector ⟨int⟩ & jm_super_row_ids, vector ⟨int⟩ & dest_disk_ids, int *dm_ids, JER_Region *region = NULL)**: For each element in **jm_super_row_ids**, this multiplies the specified super-row of matrix **jm** by the devices listed in **dm_ids**. **Dm_ids** should either be NULL, or contain **K** integers. If **dm_ids** is NULL, the first **K** devices are used. The result of multiplying **jm**'s super-row, **jm_super_row_ids**[i], by **K** devices is placed in the device with an id of **dest_disk_ids**[i]. The number of elements in **jm_super_row_ids** and **dest_dist_ids** must match. When a one is encountered in the matrix **jm**, the proper XOR/copy operation is performed. Otherwise, the operation is multiplication by the matrix element in $GF(2^w)$ and an XOR into the destination.

  If the optional **region** argument is included, the **Dotprod** routine may not operate on entire disks.

  If **NumberOfCores** is ≤ 1, this procedure is not multi-threaded, and each destination disk is processed separately. If **NumberOfCores** > 1, this will create multiple threads, and split the work among them. The exact number of threads and method for dividing the work depends upon the value of **MultiThreadMethod**.

  Returns 0 on success, -1 on failure.
- **void Remove_Drive(int drive)**: Deletes the device with an id of **drive**. The drive's entries in **Ptrs** and **States** are erased. The generator matrix is modified to work with **N**-1 devices.
- **void Remove_Drive_And_Re_Encode(int drive)**: This simply calls **Remove_Drive(drive)** then **Encode()**.
- **JER_Matrix * Make_Decoding_Matrix(int *dm_ids)**: This does not decode, but instead creates the decoding matrix. Note that **dm_ids** should be allocated to hold **K** integers. This procedure, will fill **dm_ids** with the ids of the first **K** disks with states of *up*. Returns NULL on failure.
- **void Do_Parity(unsigned char * parity_ptr)**: This calculates the parity of the first **K** devices in **Ptrs** and puts the result into the memory pointed to by **parity_ptr**. It assumes that **parity_ptr** has been allocated by the user to a size of **PacketSize** * **PacketsPerSlice** bytes.

7.3 Generator matrices and scheduling - jer_gen_t.cpp

The JER_Gen_T class handles the generator matrix and scheduling. Procedures in **reed_sol.cpp** and **bitmatrices.cpp** are used to create generators for various coding schemes. The variables that are members of this class are as follows:

- **int K**: Identical to **K** in **JER_Slices**, number of data drives.
- **int N**: Identical to **N** in **JER_Slices**, total number of drives.
- **JER_Matrix * M**: The coding matrix. For systematic codes, the matrix is of size (**K***WPD) X ((**N-K**)***WPD**). Otherwise, the dimensions are (**K***WPD)X(**N***WPD).
- **int WPD**: Words per drive. Indicates how many rows/columns of **M** are to be grouped together and considered as a super-row/column. Generally used when dealing with bitmatrices.
- **bool Systematic**: Whether or not the generator represents a systematic code. This is set for the user in the generator creation functions.
- **bool PDrive**: Parity drive. This is set for the user in the generator creation functions. Whether or not the first row of the coding matrix is all ones. When the coding matrix is a bitmatrix, this property should be true when the first **WPD** rows compose **K** identity matrices. When (**N-K**) $>$ 1 and the first row of the coding matrix is composed of all ones, then there are times when we can improve the performance of decoding by not following the methodology described in Section 3. This is true when coding device zero is one of the survivors, and more than one data device has been erased. In this case, it is better to decode all but one of the data devices as described in Section 3, but decode the last data device using the other data devices and coding device zero. For this reason, **PDrive** should be true if the first row of the coding matrix is all ones.
- **bool rs_r6**: Whether or not the generator matrix is a Reed-Solomon RAID 6 coding matrix. This property defaults to false, but it is set to true in **RS_R6_Generator()** (located in reed_sol.cpp). If **rs_r6** is true, **Encode()** will use Anvin's optimization [Anv07].
- **map $\langle$string, JER_Schedule *$\rangle$ Schedules**: Cache containing encode and decode schedules. The map is keyed on strings of **N** 1's and 0's. The ith character represents the state of the ith drive (0 = *up* and 1 = *down*).

The following methods are members of the JER_Gen_T class.

- **$\sim$JER_Gen_T()**: The class deconstructor simply calls the **Delete_Schedules()** method explained below.
- **int Am_I_MDS()**: Determines if the coding matrix, **M**, represents a maximum distance separable (MDS) code. The generator is MDS if **M** is invertible when any combinations of (**N-K**) rows are removed. Returns 1 if the matrix is MDS, 0 if it is not MDS, and -1 on failure.
- **int Create_Encode_Schedule(int smart)**: This method creates a schedule for encoding. If **smart** = 1, the newly created schedule's type is CSHR. The new schedule is inserted into the **Schedules** map. If the map already contains an encode schedule, the old schedule will be deleted. If **M** is not a bitmatrix (**M**$\rightarrow$**W**$\neq$ 1), nothing is done and **Create_Encode_Schedule()** returns failed. Returns 0 on success, -1 on failure.

- **int Create_R6_Schedules(int smart)**: First, this method deletes all schedules in **Schedules**. It then generates the encoding schedule, and schedules for every combination of single and double-disk erasure decoding. If **smart** = 1, it creates CSHR schedules. If **M** is not a bitmatrix (**M**→**W**≠ 1) or (**N**-**K**)≠ 2, nothing is done and **Create_R6_Schedules()** returns failed. Returns 0 on success, -1 on failure.

- **void Delete_Schedules()**: This method calls delete on all of the schedules in **Schedules**. The **Schedules** map is then emptied. This is called for the user in **JER_Gen_T**'s deconstructor.

- **JER_Schedule * Create_Single_Decode_Schedule(vector ⟨int⟩ & erased, int smart)**: Returns a schedule for decoding given a set of failed drives. If **M** is not a bitmatrix (**M**→**W**≠ 1), nothing is done and returns failed. Returns NULL on failure.

- **JER_Gen_T *Genmatrix_To_Genbitmatrix()**: This method is called on a generator whose coding matrix has **W** ≠ 1. It returns a new generator whose coding matrix has **W** = 1. The newly returned generator's properties, including **WPD**, are updated accordingly. Returns NULL on failure.

- **JER_Gen_T *Genbitmatrix_To_Genmatrix()**: This method is called on a generator whose coding matrix has **W** = 1. It returns a new generator whose coding matrix has **W** = **WPD**. The newly returned generator's properties, including **WPD**, are updated accordingly. Returns NULL on failure.

- **int Genmatrix_To_Genbitmatrix(JER_Gen_T &bgen)**: This method is called on a matrix generator, and passed a **JER_Gen_T** object (**bgen**). **Bgen** is converted to a bitmatrix representation of the current generator. Returns 0 on success, -1 on failure.

- **int Genbitmatrix_To_Genmatrix(JER_Gen_T &gen)**: This method is called on a bitmatrix generator. The argument, **gen**, is converted to a matrix representation of the current generator. Returns 0 on success, -1 on failure.

7.4 Matrices and their basic operations - jer_matrix.cpp

The JER_Matrix class handles all objects and operations associated with matrices and matrix manipulation. The JER_Matrix class contains the following variables:

- **int R**: The number of rows the matrix contains.

- **int C**: The number of columns the matrix contains.

- **int W**: All elements of the matrix will be elements of GF(2^W) and the galois field will be used in all operations concerning this matrix.

- **vector⟨uint64_t⟩ Elts**: The container in which the elements reside. The packing of the vector is dependent upon **W**. For bitmatrices (**W** = 1), each number in the matrix is stored as a single bit. In matrices with 2≤**W**≤8, numbers are stored as 1 byte. In matrices with 9≤**W**≤16, numbers are stored as 2 bytes. In matrices with 17≤**W**≤32, numbers are stored as 4 bytes.

The following lists the methods of the JER_Matrix class. Any mention of the matrix refers to the JER_Matrix object through which the method is being called.

- **JER_Matrix()**: Constructs a matrix object, but does not initialize any variables.

- **JER_Matrix(int r, int c, int w)**: Constructs a matrix object, initializes **R**, **C**, and **W** to the specified values, and resizes **Elts** appropriately. **Elts** contains all zeros after resizing.
- **int Create_Empty(int r, int c, int w)**: Sets **R**, **C**, and **W** accordingly. Clears the values in **Elts** and resizes the vector to be of appropriate size. **Elts** contains all zeros after resizing. Return 0 on success, -1 on failure.
- **int Create_Identity(int r, int c, int w)**: Sets **R**, **C**, and **W** accordingly. Resizes **Elts** appropriately and sets the values on the diagonal to ones in order to create an identity matrix. If **r**≠**c**, nothing is done and **Create_Identity()** returns failed. Returns 0 on success, -1 on failure.
- **void Set(int r, int c, uint64_t val)**: Sets the **r**,**c** element of the matrix to **val**. Behavior is not specified if **val**$\notin GF(2^W)$.
- **uint64_t Get(int r, int c)**: Returns the value of the **r**,**c** element.
- **void Print()**: Prints the matrix to stdout.
- **void Print(int WPD)**: Prints the matrix to stdout. Adds extra padding every **WPD** rows and columns to help distinguish super-rows and super-columns in the case of a bitmatrix.
- **string String()**: Returns a string of the contents of the matrix.
- **string String(int WPD)**: Returns a string of the contents of the matrix. Adds extra padding every **WPD** rows and columns to help distinguish super rows and super columns in the case of a bitmatrix.
- **int Copy(JER_Matrix* m)**: Sets **R**, **C**, and **W** to that of **m**. Copies the elements of **m** into the matrix, resizing **Elts** in the process. Returns 0 on success, -1 on failure.
- **int Add(JER_Matrix *m)**: Xors each element of the matrix with its corresponding element in **m**, stores the result in the matrix. If **R**, **C**, **W** $\neq$ **m**$\rightarrow$ **R**, **m**$\rightarrow$ **C**, **m**$\rightarrow$ **W**, nothing is done and **Add()** returns failed. Returns 0 on success, -1 on failure.
- **int Append_Row(JER_Matrix *m, int r)**: Appends the rth row of m to the matrix. If **C**, **W** $\neq$ **m**$\rightarrow$ **C**, **m**$\rightarrow$ **W**, nothing is done and **Append_Row()** returns failed. Returns 0 on success, -1 on failure.
- **int Append_Col(JER_Matrix *m, int c)**: Appends the **c**th column of **m** to the matrix. If **R**, **W** $\neq$ **m**$\rightarrow$ **R**, **m**$\rightarrow$ **W**, nothing is done and **Append_Col()** returns failed. Returns 0 on success, -1 on failure.
- **int Delete_Row(int r)**: Deletes the **r**th row of the matrix. Resizes **Elts** accordingly. Returns 0 on success, -1 on failure.
- **int Delete_Col(int c)**: Deletes the **c**th column of the matrix. Resizes **Elts** accordingly. Returns 0 on success, -1 on failure.
- **int Delete_Rows(int starting_r, int nr)**: Deletes **nr** rows of the matrix, beginning with row **starting_r**. Resizes **Elts** accordingly. Returns 0 on success -1 on failure.
- **int Delete_Cols(int starting_c, int nc)**: Deletes **nc** columns of the matrix, beginning with column **starting_c**. Resizes **Elts** accordingly. Returns 0 on success -1 on failure.

- **int Copy_Panel(JER_Matrix *m, int dest_r, int dest_c)**: Copies **m** into the matrix placing **m**[0][0] at matrix[**dest_r**][**dest_c**]. If **R** < **m**→**R** + **dest_r** or **C** < **m**→**C** + **dest_c** or **W** ≠ **m**→**W**, nothing is done and **Copy_Panel()** returns failed. Returns 0 on success, -1 on failure.

- **int Copy_Panel(JER_Matrix *m, int src_r, int src_c,int dest_r, int dest_c, int nr, int nc)**: Copies an **nr** by **nc** portion of **m** into the matrix placing **m**[**src_r**][**src_c**] at matrix[**dest_r**][**dest_c**]. If **R** < **nr** + **dest_r** or **C** < **nc** + **dest_c** or **m**→**R** < **nr** + **src_r** or **m**→**C** < **nc** + **src_c** or **W** ≠ **m**→**W**, nothing is done and **Copy_Panel()** returns failed. Returns 0 on success, -1 on failure.

- **int Add_Panel(JER_Matrix *m, int dest_r, int dest_c)**: Adds **m** into the matrix xorring **m**[0][0] with matrix[**dest_r**][**dest_c**]. If **R** < **m**→**R** + **dest_r** or **C** < **m**→**C** + **dest_c** or **W** ≠ **m**→**W**, nothing is done and **Add_Panel()** returns failed. Returns 0 on success, -1 on failure.

- **int Add_Panel(JER_Matrix *m, int src_r, int src_c,int dest_r, int dest_c, int nr, int nc)**: Adds an **nr** by **nc** portion of **m** into the matrix xorring **m**[**src_r**][**src_c**] with matrix[**dest_r**][**dest_c**]. If **R** < **nr** + **dest_r** or **C** < **nc** + **dest_c** or **m**→**R** < **nr** + **src_r** or **m**→**C** < **nc** + **src_c** or **W** ≠ **m**→**W**, nothing is done and **Add_Panel()** returns failed. Returns 0 on success, -1 on failure.

- **void Horizontal_Rotate(int cols)**: Shifts each column of the matrix left by **cols**, wrapping around those columns who would go out of range. If **cols** is negative, the columns are shifted right.

- **void Vertical_Rotate(int rows)**: Shifts each row of the matrix up by **rows**, wrapping around those rows who would go out of range. If **rows** is negative, the rows are shifted down.

- **void Swap_Rows(int r1, int r2)**: Swaps rows **r1** and **r2** of the matrix.

- **void Swap_Cols(int c1, int c2)**: Swaps columns **c1** and **c2** of the matrix.

- **void Row_PlusEquals(int r1, int r2)**: Stores in row **r1** the xor of rows **r1** and **r2**.

- **void Col_PlusEquals(int c1, int c2)**: Stores in column **c1** the xor of columns **c1** and **c2**.

- **void Row_PlusEquals_Prod(int r1, int r2, int prod)**: Stores in row **r1** the xor of rows **r1** and (**r2** * **prod**) in $GF(2^W)$.

- **void Col_PlusEquals_Prod(int c1, int c2, int prod)**: Stores in column **c1** the xor of column **c1** and (**c2** * **prod**) in $GF(2^W)$.

- **void Row_TimesEquals(int r1, int prod)**: Multiplies row **r1** of the matrix by **prod** in $GF(2^W)$.

- **void Col_TimesEquals(int c1, int prod)**: Multiplies column **c1** of the matrix by **prod** in $GF(2^W)$.

- **JER_Schedule *To_Schedule_CSHR(int WPD)**: Creates and returns a schedule based off of the matrix. **WPD** indicates how many rows/columns form a super-row/column. If **W** ≠ 1 or **C**,**R** = 0, nothing is done and **To_Schedule_CSHR()** returns failed. Returns NULL on failure. Uses Code Specific Hybrid Reconstruction to generate the schedule.

- **JER_Schedule *To_Schedule_Dumb(int WPD)**: Creates and returns a schedule based off of the matrix. **WPD** indicates how many rows/columns form a super-row/column. If **W** ≠ 1 or **C**,**R** = 0, nothing is done and **To_Schedule_Dumb()** returns failed. Returns NULL on failure.

- **JER_Matrix *Matrix_To_Bitmatrix()**: Creates and returns a matrix that is the bitmatrix representation of the original matrix. The returned matrix has $\mathbf{W} = 1$. Returns NULL on failure.
- **JER_Matrix *Bitmatrix_To_Matrix(int WPD)**: Creates and returns a matrix that is the matrix representation of the original bitmatrix. **WPD** indicates the galois field into which to convert the bitmatrix. The returned matrix has $\mathbf{W} = \mathbf{WPD}$. If $\mathbf{W} \neq 1$, nothing is done and **Bitmatrix_To_Matrix()** returns failed. Returns NULL on failure.
- **int Matrix_To_Bitmatrix(JER_Matrix &jbm)**: Modifies **jbm** to be the bitmatrix representation of the original matrix. After conversion, $\mathbf{jbm}\rightarrow\mathbf{W} = 1$. Returns 0 on success, -1 on failure.
- **int Bitmatrix_To_Matrix(JER_Matrix &jm, int WPD)**: Modifies **jm** to be the matrix representation of the original bitmatrix. **WPD** indicates the galois field into which to convert the bitmatrix. After conversion, $\mathbf{jm}\rightarrow\mathbf{W} = \mathbf{WPD}$. If $\mathbf{W} \neq 1$, nothing is done and **Bitmatrix_To_Matrix()** returns failed. Returns 0 on success, -1 on failure.

These next methods do not belong to the JER_Matrix class, but are stongly associated with the class and their specifications can be found in jer_matrix.cpp.

- **JER_Matrix *Sum(JER_Matrix *m1, JER_Matrix *m2)**: Creates and returns a matrix that is the xor of the two provided matrices. If $\mathbf{m1}\rightarrow\mathbf{R} \neq \mathbf{m2}\rightarrow\mathbf{R}$, $\mathbf{m1}\rightarrow\mathbf{C} \neq \mathbf{m2}\rightarrow\mathbf{C}$, $\mathbf{m1}\rightarrow\mathbf{W} \neq \mathbf{m2}\rightarrow\mathbf{W}$, nothing is done and **Sum()** returns failed. Returns NULL on failure.
- **JER_Matrix Sum(JER_Matrix &m1, JER_Matrix &m2)**: Creates and returns a matrix that is the xor of the two provided matrices. If $\mathbf{m1}.\mathbf{R} \neq \mathbf{m2}.\mathbf{R}$, $\mathbf{m1}.\mathbf{C} \neq \mathbf{m2}.\mathbf{C}$, $\mathbf{m1}.\mathbf{W} \neq \mathbf{m2}.\mathbf{W}$, nothing is done and **Sum()** returns an empty matrix ($\mathbf{R}$=0 && $\mathbf{C}$=0).
- **int Sum(JER_Matrix *m1, JER_Matrix *m2, JER_Matrix *sum)**: Modifies **sum** to be the xor of the two provided matrices. If $\mathbf{m1}\rightarrow\mathbf{R} \neq \mathbf{m2}\rightarrow\mathbf{R}$, $\mathbf{m1}\rightarrow\mathbf{C} \neq \mathbf{m2}\rightarrow\mathbf{C}$, $\mathbf{m1}\rightarrow\mathbf{W} \neq \mathbf{m2}\rightarrow\mathbf{W}$, nothing is done and **Sum()** returns failed. Returns 0 on success, -1 on failure.
- **int Sum(JER_Matrix &m1, JER_Matrix &m2, JER_Matrix &sum)**: Modifies **sum** to be the xor of the two provided matrices. If $\mathbf{m1}.\mathbf{R} \neq \mathbf{m2}.\mathbf{R}$, $\mathbf{m1}.\mathbf{C} \neq \mathbf{m2}.\mathbf{C}$, $\mathbf{m1}.\mathbf{W} \neq \mathbf{m2}.\mathbf{W}$, nothing is done and **Sum()** returns failed. Returns 0 on success, -1 on failure.
- **JER_Matrix *Prod(JER_Matrix *m1, JER_Matrix *m2)**: Creates and returns a matrix that is the product of the two provided matrices in $GF(2^W)$. If $\mathbf{m1}\rightarrow\mathbf{C} \neq \mathbf{m2}\rightarrow\mathbf{R}$ or $\mathbf{m1}\rightarrow\mathbf{W} \neq \mathbf{m2}\rightarrow\mathbf{W}$, nothing is done and **Prod()** returns failed. Returns NULL on failure.
- **JER_Matrix Prod(JER_Matrix &m1, JER_Matrix &m2)**: Creates and returns a matrix that is the product of the two provided matrices in $GF(2^W)$. If $\mathbf{m1}.\mathbf{C} \neq \mathbf{m2}.\mathbf{R}$ or $\mathbf{m1}.\mathbf{W} \neq \mathbf{m2}.\mathbf{W}$, nothing is done and **Prod()** returns failed. An empty matrix ($\mathbf{R}$=0 && $\mathbf{C}$=0) is returned on failure.
- **int Prod(JER_Matrix *m1, JER_Matrix *m2, JER_Matrix *prod)**: Modifies **prod** to be the product of the two provided matrices in $GF(2^W)$. If $\mathbf{m1}\rightarrow\mathbf{C} \neq \mathbf{m2}\rightarrow\mathbf{R}$ or $\mathbf{m1}\rightarrow\mathbf{W} \neq \mathbf{m2}\rightarrow\mathbf{W}$, nothing is done and **Prod()** returns failed. Returns 0 on success, -1 on failure.
- **int Prod(JER_Matrix &m1, JER_Matrix &m2, JER_Matrix &prod)**: Modifies **prod** to be the product of the two provided matrices in $GF(2^W)$. If $\mathbf{m1}.\mathbf{C} \neq \mathbf{m2}.\mathbf{R}$ or $\mathbf{m1}.\mathbf{W} \neq \mathbf{m2}.\mathbf{W}$, nothing is done and **Prod()** returns failed. Returns 0 on success, -1 on failure.

- **JER_Matrix *Inverse(JER_Matrix *m1)**: Creates and returns a matrix that is the inverse of **m1**. If **m1** is not invertible, nothing is done and **Inverse()** returns failed. Returns NULL on failure.
- **JER_Matrix Inverse(JER_Matrix *m1)**: Creates and returns a matrix that is the inverse of **m1**. If **m1** is not invertible, nothing is done and **Inverse()** returns an empty matrix (**R**=0 && **C**=0).
- **int Inverse(JER_Matrix *m1, JER_Matrix *inv)**: Modifies **inv** to be the inverse of **m1**. If **m1** is not invertible, **inv** is emptied (**inv**→**R**=0 && **inv**→**C**=0), and **Inverse()** returns failure. Returns 0 on success, -1 on failure.
- **int Inverse(JER_Matrix &m1, JER_Matrix &inv)**: Modifies **inv** to be the inverse of **m1**. If **m1** is not invertible, **inv** is emptied (**inv**.**R**=0 && **inv**.**C**=0), and **Inverse()** returns failure. Returns 0 on success, -1 on failure.

7.5 Example programs

The following programs reside in the **Examples** directory. These demonstrate the usage of the classes contained in **jerasure.h**. They are as follows:

- **jerasure_01.cpp**: This takes three parameters: r, c and w. It creates an $r \times c$ matrix in $GF(2^w)$, where the element in row i, column j is equal to 2^{ci+j} in $GF(2^w)$. Rows and columns are zero-indexed. Example:

```
UNIX> jerasure_01 3 15 8
  1   2   4   8  16  32  64 128  29  58 116 232 205 135  19
 38  76 152  45  90 180 117 234 201 143   3   6  12  24  48
 96 192 157  39  78 156  37  74 148  53 106 212 181 119 238
UNIX>
```

  This demonstrates usage of **galois_single_multiply()**, **JER_Matrix::Set()**, and **JER_Matrix::Print()**.

- **jerasure_02.cpp**: This takes three parameters: r, c and w. It creates the same matrix as in **jerasure_01**, and then converts it to a $rw \times cw$ bit-matrix and prints it out. Example:

```
UNIX> jerasure_02 3 10 4
1000 0001 0010 0100 1001 0011 0110 1101 1010 0101
0100 1001 0011 0110 1101 1010 0101 1011 0111 1111
0010 0100 1001 0011 0110 1101 1010 0101 1011 0111
0001 0010 0100 1001 0011 0110 1101 1010 0101 1011

1011 0111 1111 1110 1100 1000 0001 0010 0100 1001
1110 1100 1000 0001 0010 0100 1001 0011 0110 1101
1111 1110 1100 1000 0001 0010 0100 1001 0011 0110
0111 1111 1110 1100 1000 0001 0010 0100 1001 0011

0011 0110 1101 1010 0101 1011 0111 1111 1110 1100
1010 0101 1011 0111 1111 1110 1100 1000 0001 0010
1101 1010 0101 1011 0111 1111 1110 1100 1000 0001
0110 1101 1010 0101 1011 0111 1111 1110 1100 1000
UNIX>
```

  This demonstrates usage of **galois_single_multiply()**, **JER_Matrix::Set()**, **JER_Matrix::Matrix_To_Bitmatrix()**, and **JER_Matrix::Print()**.

- **jerasure_03.cpp**: This takes three parameters: k and w. It creates a $k \times k$ Cauchy matrix in $GF(2^w)$, and tests invertibility.

  The parameter k must be less than 2^w. The element in row i, column j is set to:

  $$\frac{1}{i \oplus (2^w - j - 1)}$$

  where division is in $GF(2^w)$, $\oplus$ is XOR and subtraction is regular integer subtraction. When $k > 2^{w-1}$, there will be i and j such that $i \oplus (2^w - j - 1) = 0$. When that happens, we set that matrix element to zero.

  After creating the matrix and printing it, we test whether it is invertible. If $k \leq 2^{w-1}$, then it will be invertible. Otherwise it will not. Then, if it is invertible, it prints the inverse, then multplies the inverse by the original matrix and prints the product which is the identity matrix. Examples:

```
UNIX> jerasure_03 4 3
The Cauchy Matrix:
4 3 2 7
3 4 7 2
2 7 4 3
7 2 3 4

Invertible: Yes

Inverse:
1 2 5 3
2 1 3 5
5 3 1 2
3 5 2 1

Inverse times matrix (should be identity):
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1

UNIX>

UNIX> jerasure_03 5 3
The Cauchy Matrix:
4 3 2 7 6
3 4 7 2 5
2 7 4 3 1
7 2 3 4 0
6 5 1 0 4

Invertible: No

UNIX>
```

  This demonstrates usage of **galois_single_divide()**, **JER_Matrix::Set()**, **JER_Matrix::Print()**, **Inverse()**, and **Prod()**.

- **jerasure_04.cpp**: This does the exact same thing as **jerasure_03**, except it uses **JER_Matrix::Matrix_To_Bitmatrix()** to convert the Cauchy matrix to a bit-matrix, and then uses the bit-matrix operations to test invertibility and to invert the matrix. Examples:

```
UNIX> jerasure_04 4 3
The Cauchy Bit-Matrix:
010 101 001 111
011 111 101 100
101 011 010 110

101 010 111 001
111 011 100 101
011 101 110 010

001 111 010 101
101 100 011 111
010 110 101 011

111 001 101 010
100 101 111 011
110 010 011 101

Invertible: Yes

Inverse:
100 001 110 101
010 101 001 111
001 010 100 011

001 100 101 110
101 010 111 001
010 001 011 100

110 101 100 001
001 111 010 101
100 011 001 010

101 110 001 100
111 001 101 010
011 100 010 001

Inverse times matrix (should be identity):
100 000 000 000
010 000 000 000
001 000 000 000

000 100 000 000
000 010 000 000
000 001 000 000

000 000 100 000
000 000 010 000
000 000 001 000

000 000 000 100
000 000 000 010
000 000 000 001
UNIX>

UNIX> jerasure_04 5 3
The Cauchy Bit-Matrix:
```

```
010 101 001 111 011
011 111 101 100 110
101 011 010 110 111

101 010 111 001 110
111 011 100 101 001
011 101 110 010 100

001 111 010 101 100
101 100 011 111 010
010 110 101 011 001

111 001 101 010 000
100 101 111 011 000
110 010 011 101 000

011 110 100 000 010
110 001 010 000 011
111 100 001 000 101

Invertible: No

UNIX>
```

This demonstrates usage of **galois_single_divide()**, **JER_Matrix::Set()**, **JER_Matrix::Matrix_-To_Bitmatrix()**, **JER_Matrix::Print()**, **Inverse()**, and **Prod()**.

8 Part 3 of the Library: Reed-Solomon Coding (reed_sol.h)

The files **Reed_sol.h** and **reed_sol.cpp** implement procedures that are specific to Vandermonde matrix-based Reed-Solomon coding, Cauchy matrix-based Reed-Solomon coding, and for Reed-Solomon coding optimized for RAID-6. Refer to [Pla97, PD05] for a description of classic Reed-Solomon coding and to [Anv07] for Reed-Solomon coding optimized for RAID-6. Methods beginning with RS refer to Vandermonde based codings. The methods beginning with CRS are Cauchy based, and those beginning with RS_R6 are opitimized for RAID-6. Where not specified, the parameters are as described in Section 7.

8.1 Cauchy matrices

We don't use the Cauchy matrices described in [PX06], because there is a simple heuristic that creates better matrices:

- Construct the usual Cauchy matrix M such that $M[i,j] = \frac{1}{i \oplus (m+j)}$, where division is over $GF(2^w)$, $\oplus$ is XOR and the addition is regular integer addition.
- For each column j, divide each element (in $GF(2^w)$) by $M[0,j]$. This has the effect of turning each element in row 0 to one.
- Next, for each row $i > 0$ of the matrix, do the following:
  - Count the number of ones in the bit representation of the row.

- Count the number of ones in the bit representation of the row divided by element $M[i,j]$ for each j.
- Whichever value of j gives the minimal number of ones, if it improves the number of ones in the original row, divide row i by $M[i,j]$.

While this does not guarantee an optimal number of ones, it typically generates a good matrix. For example, suppose $k = m = w = 3$. The matrix M is as follows:

$$\begin{vmatrix} 6 & 7 & 2 \\ 5 & 2 & 7 \\ 1 & 3 & 4 \end{vmatrix}$$

First, we divide column 0 by 6, column 1 by 7 and column 2 by 2, to yield:

$$\begin{vmatrix} 1 & 1 & 1 \\ 4 & 3 & 6 \\ 3 & 7 & 2 \end{vmatrix}$$

Now, we concentrate on row 1. Its bitmatrix representation has 5+7+7 = 19 ones. If we divide it by 4, the bitmatrix has 3+4+5 = 12 ones. If we divide it by 3, the bitmatrix has 4+3+4 = 11 ones. If we divide it by 6, the bitmatrix has 6+7+3 = 16 ones. So, we replace row 1 with row 1 divided by 3.

We do the same with row 2 and find that it will have the minimal number of ones when it is divided by three. The final matrix is:

$$\begin{vmatrix} 1 & 1 & 1 \\ 5 & 1 & 2 \\ 1 & 4 & 7 \end{vmatrix}$$

This matrix has 34 ones, a distinct improvement over the original matrix that has 46 ones. The best matrix in [PX06] has 39 ones. This is because the authors simply find the best X and Y, and do not modify the matrix after creating it.

8.2 Reed-Solomon generators - reed_sol.cpp

The following are the methods used to generate matrices for Reed-Solomon coding:

- **int CRS_N_Ones(int n, int w)**: Returns the number of ones in the bitmatrix representation of the number **n**. The argument **n** must exist in $GF(2^w)$.
- **JER_Matrix *RS_Nonsystematic_Matrix(int n, int k, int w)**: Creates and returns a non-systematic Vandermonde matrix for Reed-Solomon coding. If **n**,**k** $> 2^w$, nothing is done and **RS_Nonsystematic_Matrix()** returns failed. Returns NULL on failure.
- **JER_Gen_T *RS_Nonsystematic_Generator(int n, int k, int w)**: Creates and returns a generator containing a non-systematic Vandermonde matrix for Reed-Solomon coding. If **n**,**k** $> 2^w$, nothing is done and **RS_Nonsystematic_Generator()** returns failed. Returns NULL on failure.
- **int RS_Nonsystematic_Generator(int n, int k, int w, JER_Gen_T &g)**: Modifies **g** to be a generator containing a non-systematic Vandermonde matrix for Reed-Solomon coding. If **n**,**k** $> 2^w$, nothing is done and **RS_Nonsystematic_Generator()** returns failed. Return 0 on success, -1 on failure.

- **JER_Matrix *RS_Extended_Matrix(int n, int k, int w)**: Creates and returns an extended systematic Vandermonde matrix for Reed-Solomon coding. If **n**,**k** $> 2^w$, nothing is done and **RS_Extended_Matrix()** returns failed. Returns NULL on failure.

- **JER_Gen_T *RS_Extended_Generator(int n, int k, int w)**: Creates and returns a generator containing an extended Vandermonde matrix for Reed-Solomon coding. If **n**,**k** $> 2^w$, nothing is done and **RS_Extended_Generator()** returns failed. Returns NULL on failure.

- **int RS_Extended_Generator(int n, int k, int w, JER_Gen_T &g)**: Modifies **g** to be a generator containing an extended Vandermonde matrix for Reed-Solomon coding. If **n**,**k** $> 2^w$, nothing is done and **RS_Extended_Generator()** returns failed. Return 0 on success, -1 on failure.

- **JER_Matrix *RS_Classic_Matrix(int n, int k, int w)**: Creates and returns an extended systematic Vandermonde matrix for Reed-Solomon coding. If **n**,**k** $> 2^w$, nothing is done and **RS_Classic_Matrix()** returns failed. Returns NULL on failure.

- **JER_Gen_T *RS_Classic_Generator(int n, int k, int w)**: Creates and returns a generator containing a Vandermonde matrix for Reed-Solomon coding. If **n**,**k** $> 2^w$, nothing is done and **RS_Classic_Generator()** returns failed. Returns NULL on failure.

- **int RS_Classic_Generator(int n, int k, int w, JER_Gen_T &g)**: Modifies **g** to be a generator containing a Vandermonde matrix for Reed-Solomon coding. If **n**,**k** $> 2^w$, nothing is done and **RS_Classic_Generator()** returns failed. Return 0 on success, -1 on failure.

- **JER_Gen_T *RS_R6_Generator(int k, int w)**: Creates and returns a generator containing a matrix for RAID-6 Reed-Solomon coding. If **n**,**k** $> 2^w$, nothing is done and **RS_R6_Generator()** returns failed. Returns NULL on failure.

- **int RS_R6_Generator(int k, int w, JER_Gen_T &g)**: Modifies **g** to be a generator containing a matrix for RAID-6 Reed-Solomon coding. If **n**,**k** $> 2^w$, nothing is done and **RS_R6_Generator()** returns failed. Return 0 on success, -1 on failure.

- **JER_Gen_T *CRS_Generator(int k, int m, int w)**: Creates and returns a generator containing a Cauchy matrix for Reed-Solomon coding. If **m** + **k** $> 2^w$, nothing is done and **CRS_Generator()** returns failed. Returns NULL on failure.

- **int CRS_Generator(int k, int m, int w, JER_Gen_T &g)**: Modifies **g** to be a generator containing a Cauchy matrix for Reed-Solomon coding. If **m** + **k** $> 2^w$, nothing is done and **CRS_Generator()** returns failed. Return 0 on success, -1 on failure.

- **JER_Gen_T *CRS_XY_Generator(int k, int m, int w, vector ⟨int⟩ &X, vector ⟨int⟩ &Y)**: Creates and returns a generator containing a Cauchy matrix created based off of the **X** and **Y** sets for Reed-Solomon coding. If **m** + **k** $> 2^w$, nothing is done and **CRS_XY_Generator()** returns failed. Returns NULL on failure.

- **int CRS_XY_Generator(int k, int m, int w, vector ⟨int⟩ &X, vector ⟨int⟩ &Y, JER_Gen_T &g)**: Modifies **g** to be a generator containing a Cauchy matrix based off of the **X** and **Y** sets matrix for Reed-Solomon coding. If **m** + **k** $> 2^w$, nothing is done and **CRS_XY_Generator()** returns failed. Return 0 on success, -1 on failure.

- **int CRS_Improve_Generator(JER_Gen_T *g)**: Row reduces the matrix held by **g** to minimize the number of ones in its bitmatrix representation.
- **JER_Gen_T *CRS_Good_Generator_Bitmatrix(int k, int m, int w)**: Creates and returns a generator containing a Cauchy bitmatrix that has been improved. For $\mathbf{m} = 2$, the function returns an optimal bitmatrix. If $\mathbf{m} + \mathbf{k} > 2^w$, nothing is done and **CRS_Good_Generator_Bitmatrix()** returns failed. Returns NULL on failure.
- **int CRS_Good_Generator_Bitmatrix(int k, int m, int w, JER_Gen_T &g)**: Modifies **g** to be a generator containing a Cauchy bitmatrix that has been improved. For $\mathbf{m} = 2$, the function returns an optimal bitmatrix. If $\mathbf{m} + \mathbf{k} > 2^w$, nothing is done and **CRS_Good_Generator_Bitmatrix()** returns failed. Returns 0 on success, -1 on failure.

8.3 Example Programs

There are eleven example programs to demonstrate the use of the procedures in **reed_sol.cpp**.

- **reed_sol_rs_01.cpp**: This takes three parameters: k, m and w. It performs a classic Reed-Solomon coding of k devices onto m devices, using a Vandermonde-based distribution matrix in $GF(2^w)$. w must be 8, 16 or 32. Each device is set up to hold **8** bytes. It uses **RS_Classic_Generator()** to generate the distribution matrix, and then procedures from **jer_slices.cpp** to perform the coding and decoding.
  Example:

```
UNIX> reed_sol_rs_01 7 7 8
Last m rows of the Distribution Matrix:

  1   1   1   1   1   1   1
  1 199 210 240 105 121 248
  1  70  91 245  56 142 167
  1 170 114  42  87  78 231
  1  38 236  53 233 175  65
  1  64 174 232  52 237  39
  1 187 104 210 211 105 186

Original data:

Data                                Coding
D0  (up): c0 c9 fc 5f 6e b1 dd 15   C0  (down): 00 00 00 00 00 00 00 00
D1  (up): 91 67 6b 6f 0a e8 55 0c   C1  (down): 00 00 00 00 00 00 00 00
D2  (up): f2 11 95 64 d0 14 e5 49   C2  (down): 00 00 00 00 00 00 00 00
D3  (up): ae bb 33 2f 69 d1 99 58   C3  (down): 00 00 00 00 00 00 00 00
D4  (up): b4 46 5f 5f ba 16 dc 6f   C4  (down): 00 00 00 00 00 00 00 00
D5  (up): 3b f7 46 2d 48 08 18 39   C5  (down): 00 00 00 00 00 00 00 00
D6  (up): 5c f4 4e 21 0b 34 c3 5d   C6  (down): 00 00 00 00 00 00 00 00

Encoding complete:

Data                                Coding
D0  (up): c0 c9 fc 5f 6e b1 dd 15   C0  (up): de 41 66 28 24 b6 f3 03
D1  (up): 91 67 6b 6f 0a e8 55 0c   C1  (up): 94 f8 b5 5e 55 94 b7 72
D2  (up): f2 11 95 64 d0 14 e5 49   C2  (up): 39 7b 52 ac 33 3b 2a 29
D3  (up): ae bb 33 2f 69 d1 99 58   C3  (up): 6e 92 59 73 af 8e 29 78
```

```
D4  (up): b4 46 5f 5f ba 16 dc 6f    C4  (up): fc be 8f ea 1f 84 a0 a8
D5  (up): 3b f7 46 2d 48 08 18 39    C5  (up): 32 89 7b 83 5d 44 2b 33
D6  (up): 5c f4 4e 21 0b 34 c3 5d    C6  (up): 19 73 3f b1 ae 72 18 91

Erased 7 random devices:

Data                                 Coding
D0  (up  ): c0 c9 fc 5f 6e b1 dd 15  C0  (down): 00 00 00 00 00 00 00 00
D1  (down): 00 00 00 00 00 00 00 00  C1  (up  ): 94 f8 b5 5e 55 94 b7 72
D2  (down): 00 00 00 00 00 00 00 00  C2  (down): 00 00 00 00 00 00 00 00
D3  (up  ): ae bb 33 2f 69 d1 99 58  C3  (down): 00 00 00 00 00 00 00 00
D4  (up  ): b4 46 5f 5f ba 16 dc 6f  C4  (up  ): fc be 8f ea 1f 84 a0 a8
D5  (up  ): 3b f7 46 2d 48 08 18 39  C5  (up  ): 32 89 7b 83 5d 44 2b 33
D6  (down): 00 00 00 00 00 00 00 00  C6  (down): 00 00 00 00 00 00 00 00

State of the system after decoding:

Data                                 Coding
D0  (up): c0 c9 fc 5f 6e b1 dd 15    C0  (up): de 41 66 28 24 b6 f3 03
D1  (up): 91 67 6b 6f 0a e8 55 0c    C1  (up): 94 f8 b5 5e 55 94 b7 72
D2  (up): f2 11 95 64 d0 14 e5 49    C2  (up): 39 7b 52 ac 33 3b 2a 29
D3  (up): ae bb 33 2f 69 d1 99 58    C3  (up): 6e 92 59 73 af 8e 29 78
D4  (up): b4 46 5f 5f ba 16 dc 6f    C4  (up): fc be 8f ea 1f 84 a0 a8
D5  (up): 3b f7 46 2d 48 08 18 39    C5  (up): 32 89 7b 83 5d 44 2b 33
D6  (up): 5c f4 4e 21 0b 34 c3 5d    C6  (up): 19 73 3f b1 ae 72 18 91

UNIX>
```

This demonstrates usage of **RS_Classic_Generator()**, **JER_Matrix::Print()**, **JER_Slices::Encode()** and **JER_Slices::Decode()**.

- **reed_sol_rs_02.cpp**: This takes three parameters: k, m and w. It creates and prints three matrices in $GF(2^w)$:

  1. A $(k+m) \times k$ non-systematic Vandermonde Matrix.
  2. The $(k+m) \times k$ extended Vandermonde matrix created by converting the non-systematic Vandermonde matrix into one where the first k rows are an identity matrix. Then row k is converted so that it is all ones, and the first column is also converted so that it is all ones.
  3. The $m \times k$ classic Vandermonde coding matrix, which is last m rows of the above matrix. This is the matrix which is passed to the encoding/decoding procedures of **jer_slices.cpp**. Note that the first row of this matrix is all ones, so the generator's **PDrive** argument will be set to true.

  Note also that w may have any value from 1 to 32.
  Example:

```
UNIX> reed_sol_rs_02 6 4 11
Non-systematic Vandermonde Matrix:

   1    0    0    0    0    0
   1    1    1    1    1    1
   1    2    4    8   16   32
   1    3    5   15   17   51
   1    4   16   64  256 1024
```

```
   1    5   17   85  257 1285
   1    6   20  120  272 1632
   1    7   21  107  273 1911
   1    8   64  512   10   80
   0    0    0    0    0    1

Vandermonde Extended Matrix:

   1    0    0    0    0    0
   0    1    0    0    0    0
   0    0    1    0    0    0
   0    0    0    1    0    0
   0    0    0    0    1    0
   0    0    0    0    0    1
   1    1    1    1    1    1
   1 1879 1231 1283  682 1538
   1 1366 1636 1480  683  934
   1 1023 2045 1027 2044 1026

Vandermonde Classic Coding Matrix:

   1    1    1    1    1    1
   1 1879 1231 1283  682 1538
   1 1366 1636 1480  683  934
   1 1023 2045 1027 2044 1026

UNIX>
```

This demonstrates usage of **RS_Nonsystematic_Matrix()**, **RS_Extended_Matrix()**, **RS_Classic_-Matrix()**, and **JER_Matrix::Print()**.

- **reed_sol_rs_r6_01.cpp**: This takes two parameters: k and w. It performs RAID-6 coding using Anvin's optimization [Anv07] in $GF(2^w)$, where w must be 8, 16 or 32. It then decodes using **JER_-Slices::Decode()**.
  Example:

```
UNIX> reed_sol_rs_r6_01 9 8
Last 2 rows of the Distribution Matrix:

  1   1   1   1   1   1   1   1   1
  1   2   4   8  16  32  64 128  29

Original data:

Data                                Coding
D0  (up): c0 c9 fc 5f 6e b1 dd 15   C0  (down): 00 00 00 00 00 00 00 00
D1  (up): 91 67 6b 6f 0a e8 55 0c   C1  (down): 00 00 00 00 00 00 00 00
D2  (up): f2 11 95 64 d0 14 e5 49
D3  (up): ae bb 33 2f 69 d1 99 58
D4  (up): b4 46 5f 5f ba 16 dc 6f
D5  (up): 3b f7 46 2d 48 08 18 39
D6  (up): 5c f4 4e 21 0b 34 c3 5d
D7  (up): 66 da 6d 63 ea 37 78 32
D8  (up): 81 6d b3 49 83 1c 2a 6c

Encoding complete:
```

```
Data                                 Coding
D0  (up): c0 c9 fc 5f 6e b1 dd 15    C0  (up): 39 f6 b8 02 4d 9d a1 5d
D1  (up): 91 67 6b 6f 0a e8 55 0c    C1  (up): 7d a6 d5 a0 23 d8 17 ed
D2  (up): f2 11 95 64 d0 14 e5 49
D3  (up): ae bb 33 2f 69 d1 99 58
D4  (up): b4 46 5f 5f ba 16 dc 6f
D5  (up): 3b f7 46 2d 48 08 18 39
D6  (up): 5c f4 4e 21 0b 34 c3 5d
D7  (up): 66 da 6d 63 ea 37 78 32
D8  (up): 81 6d b3 49 83 1c 2a 6c

Erased 2 random devices:

Data                                   Coding
D0  (up  ): c0 c9 fc 5f 6e b1 dd 15    C0  (up): 39 f6 b8 02 4d 9d a1 5d
D1  (up  ): 91 67 6b 6f 0a e8 55 0c    C1  (up): 7d a6 d5 a0 23 d8 17 ed
D2  (down): 00 00 00 00 00 00 00 00
D3  (up  ): ae bb 33 2f 69 d1 99 58
D4  (up  ): b4 46 5f 5f ba 16 dc 6f
D5  (up  ): 3b f7 46 2d 48 08 18 39
D6  (up  ): 5c f4 4e 21 0b 34 c3 5d
D7  (up  ): 66 da 6d 63 ea 37 78 32
D8  (down): 00 00 00 00 00 00 00 00

State of the system after decoding:

Data                                 Coding
D0  (up): c0 c9 fc 5f 6e b1 dd 15    C0  (up): 39 f6 b8 02 4d 9d a1 5d
D1  (up): 91 67 6b 6f 0a e8 55 0c    C1  (up): 7d a6 d5 a0 23 d8 17 ed
D2  (up): f2 11 95 64 d0 14 e5 49
D3  (up): ae bb 33 2f 69 d1 99 58
D4  (up): b4 46 5f 5f ba 16 dc 6f
D5  (up): 3b f7 46 2d 48 08 18 39
D6  (up): 5c f4 4e 21 0b 34 c3 5d
D7  (up): 66 da 6d 63 ea 37 78 32
D8  (up): 81 6d b3 49 83 1c 2a 6c

UNIX>
```

This demonstrates usage of **RS_R6_Generator()**, **JER_Matrix::Print()**, **JER_Slices::Encode()**, and **JER_Slices::Decode()**.

- **reed_sol_rs_r6_02.cpp**: This takes two parameters: k and w, and performs a simple RAID-6 example using a schedule cache. Again, *packetsize* is **8** bytes. It sets up a RAID-6 coding matrix whose first row is composed of ones, and where the element in column j of the second row is equal to 2^j in $GF(2^w)$. It converts this to a bit-matrix and creates a smart encoding schedule and a schedule cache for decoding.

  It then sets the two coding devices as the erased devices, and encodes using the smart shedule. Next it deletes two random devices and uses the schedule cache to decode them. Finally, it deletes the first coding devices and recalculates it using **JER_Slices::Do_Parity()** to demonstrate that procedure.
  Example:

```
UNIX> reed_sol_rs_r6_02 5 3
Last (m * w) rows of the Binary Distribution Matrix:
```

```
100 100 100 100 100
010 010 010 010 010
001 001 001 001 001

100 001 010 101 011
010 101 011 111 110
001 010 101 011 111

Original data:

Data                              Coding
D0  (up): p0 :0000000000204a16    C0  (down): p0 :0000000000000000
          p1 :05542a27169c39e2                p1 :0000000000000000
          p2 :2eab49560bb0b8b4                p2 :0000000000000000
D1  (up): p0 :0bd068c93e5d224d    C1  (down): p0 :0000000000000000
          p1 :436c8d9c3a2adfda                p1 :0000000000000000
          p2 :1dd8ca796a67c4f1                p2 :0000000000000000
D2  (up): p0 :7742fa2948b62d69
          p1 :472e195d0681a900
          p2 :622ebb7e026bccdf
D3  (up): p0 :204d531e262b5331
          p1 :7020241c440a0e2a
          p2 :75cb1c4c41f9a5e1
D4  (up): p0 :67bc26ff181e279e
          p1 :7172c06f4909e153
          p2 :09d3bfba685ae502

Smart Schedule Encoding Complete: - 248 XOR'd bytes

Data                              Coding
D0  (up): p0 :0000000000204a16    C0  (up): p0 :3b63e70148fe319d
          p1 :05542a27169c39e2              p1 :00445a952734a041
          p2 :2eab49560bb0b8b4              p2 :2d459ba74a1ff079
D1  (up): p0 :0bd068c93e5d224d    C1  (up): p0 :77d1e3a32a47d566
          p1 :436c8d9c3a2adfda              p1 :0534a76a348384b6
          p2 :1dd8ca796a67c4f1              p2 :625de4e747f90edc
D2  (up): p0 :7742fa2948b62d69
          p1 :472e195d0681a900
          p2 :622ebb7e026bccdf
D3  (up): p0 :204d531e262b5331
          p1 :7020241c440a0e2a
          p2 :75cb1c4c41f9a5e1
D4  (up): p0 :67bc26ff181e279e
          p1 :7172c06f4909e153
          p2 :09d3bfba685ae502

Deleted both coding drives:

Data                              Coding
D0  (up): p0 :0000000000204a16    C0  (down): p0 :0000000000000000
          p1 :05542a27169c39e2                p1 :0000000000000000
          p2 :2eab49560bb0b8b4                p2 :0000000000000000
D1  (up): p0 :0bd068c93e5d224d    C1  (down): p0 :0000000000000000
          p1 :436c8d9c3a2adfda                p1 :0000000000000000
          p2 :1dd8ca796a67c4f1                p2 :0000000000000000
D2  (up): p0 :7742fa2948b62d69
```

```
          p1 :472e195d0681a900
          p2 :622ebb7e026bccdf
D3  (up): p0 :204d531e262b5331
          p1 :7020241c440a0e2a
          p2 :75cb1c4c41f9a5e1
D4  (up): p0 :67bc26ff181e279e
          p1 :7172c06f4909e153
          p2 :09d3bfba685ae502

Decoded using the smart decoding schedules: - 248 XOR'd bytes

Data                            Coding
D0  (up): p0 :0000000000204a16    C0  (up): p0 :3b63e70148fe319d
          p1 :05542a27169c39e2              p1 :00445a952734a041
          p2 :2eab49560bb0b8b4              p2 :2d459ba74a1ff079
D1  (up): p0 :0bd068c93e5d224d    C1  (up): p0 :77d1e3a32a47d566
          p1 :436c8d9c3a2adfda              p1 :0534a76a348384b6
          p2 :1dd8ca796a67c4f1              p2 :625de4e747f90edc
D2  (up): p0 :7742fa2948b62d69
          p1 :472e195d0681a900
          p2 :622ebb7e026bccdf
D3  (up): p0 :204d531e262b5331
          p1 :7020241c440a0e2a
          p2 :75cb1c4c41f9a5e1
D4  (up): p0 :67bc26ff181e279e
          p1 :7172c06f4909e153
          p2 :09d3bfba685ae502

Erased 2 random devices:

Data                              Coding
D0  (up  ): p0 :0000000000204a16    C0  (up): p0 :3b63e70148fe319d
            p1 :05542a27169c39e2              p1 :00445a952734a041
            p2 :2eab49560bb0b8b4              p2 :2d459ba74a1ff079
D1  (down): p0 :0000000000000000    C1  (up): p0 :77d1e3a32a47d566
            p1 :0000000000000000              p1 :0534a76a348384b6
            p2 :0000000000000000              p2 :625de4e747f90edc
D2  (up  ): p0 :7742fa2948b62d69
            p1 :472e195d0681a900
            p2 :622ebb7e026bccdf
D3  (up  ): p0 :204d531e262b5331
            p1 :7020241c440a0e2a
            p2 :75cb1c4c41f9a5e1
D4  (down): p0 :0000000000000000
            p1 :0000000000000000
            p2 :0000000000000000

Decoded using the dumb decoding schedules: - 384 XOR'd bytes

Data                            Coding
D0  (up): p0 :0000000000204a16    C0  (up): p0 :3b63e70148fe319d
          p1 :05542a27169c39e2              p1 :00445a952734a041
          p2 :2eab49560bb0b8b4              p2 :2d459ba74a1ff079
D1  (up): p0 :0bd068c93e5d224d    C1  (up): p0 :77d1e3a32a47d566
          p1 :436c8d9c3a2adfda              p1 :0534a76a348384b6
          p2 :1dd8ca796a67c4f1              p2 :625de4e747f90edc
D2  (up): p0 :7742fa2948b62d69
```

```
          p1 :472e195d0681a900
          p2 :622ebb7e026bccdf
D3  (up): p0 :204d531e262b5331
          p1 :7020241c440a0e2a
          p2 :75cb1c4c41f9a5e1
D4  (up): p0 :67bc26ff181e279e
          p1 :7172c06f4909e153
          p2 :09d3bfba685ae502

Erased first coding device:

Data                           Coding
D0  (up): p0 :0000000000204a16    C0  (down): p0 :0000000000000000
          p1 :05542a27169c39e2                p1 :0000000000000000
          p2 :2eab49560bb0b8b4                p2 :0000000000000000
D1  (up): p0 :0bd068c93e5d224d    C1  (up  ): p0 :77d1e3a32a47d566
          p1 :436c8d9c3a2adfda                p1 :0534a76a348384b6
          p2 :1dd8ca796a67c4f1                p2 :625de4e747f90edc
D2  (up): p0 :7742fa2948b62d69
          p1 :472e195d0681a900
          p2 :622ebb7e026bccdf
D3  (up): p0 :204d531e262b5331
          p1 :7020241c440a0e2a
          p2 :75cb1c4c41f9a5e1
D4  (up): p0 :67bc26ff181e279e
          p1 :7172c06f4909e153
          p2 :09d3bfba685ae502

Re-encoded coding device 0 with JER_Slices::Do_Parity() - 96 XOR'd bytes

Data                           Coding
D0  (up): p0 :0000000000204a16    C0  (down): p0 :3b63e70148fe319d
          p1 :05542a27169c39e2                p1 :00445a952734a041
          p2 :2eab49560bb0b8b4                p2 :2d459ba74a1ff079
D1  (up): p0 :0bd068c93e5d224d    C1  (up  ): p0 :77d1e3a32a47d566
          p1 :436c8d9c3a2adfda                p1 :0534a76a348384b6
          p2 :1dd8ca796a67c4f1                p2 :625de4e747f90edc
D2  (up): p0 :7742fa2948b62d69
          p1 :472e195d0681a900
          p2 :622ebb7e026bccdf
D3  (up): p0 :204d531e262b5331
          p1 :7020241c440a0e2a
          p2 :75cb1c4c41f9a5e1
D4  (up): p0 :67bc26ff181e279e
          p1 :7172c06f4909e153
          p2 :09d3bfba685ae502

UNIX>
```

This demonstrates usage of **RS_R6_Generator()**, **CRS_Convert_To_Bitmatrix()**, **JER_Matrix::Print()**, **JER_Gen_T::Create_Encode_Schedule()**, **JER_Slices::Encode()**, **JER_Slices::Add_Drive_Failure()**, **JER_Gen_T::Create_R6_Schedules()**, **JER_Slices::Decode()**, and **JER_Slices::Do_Parity()**.

- **reed_sol_crs_01.cpp**: This takes four parameters: k, m, w and $size$, and performs a classic Cauchy Reed-Solomon coding example in $GF(2^w)$. w must be either 8, 16 or 32, and the sum $k+m$ must be less than or equal to 2^w. The total number of bytes for each device is given by $size$ which must be a

multiple of **8**. It first sets up an $m \times k$ Cauchy coding matrix where element i, j is:

$$\frac{1}{i \oplus (m + j)}$$

where division is in $GF(2^w)$, $\oplus$ is XOR, and addition is standard integer addition. It prints out these m rows. The program then creates k data devices each with $size$ bytes of random data and encodes them into m coding devices using **JER_Slices::Encode()**. It prints out the data and coding in hexadecimal–one byte is represented by 2 hex digits. Next, it erases m random devices from the collection of data and coding devices, and prints the resulting state. Then it decodes the erased devices using **JER_Slices::Decode()** and prints the restored state. Next, it shows what the decoding matrix looks like when the first m devices are erased. This matrix is the inverse of the last k rows of the distribution matrix. And finally, it uses **JER_Slices::Make_Decoding_Matrix()** and **JER_Slices::Dotprod()** to show how to explicitly calculate the first data device from the others when the first m devices have been erased.

Here is an example for $w = 8$ with 3 data devices and 4 coding devices each with a size of 8 bytes:

```
UNIX> reed_sol_crs_01 3 4 8 8
The Coding Matrix (the last m rows of the Distribution Matrix):

 71 167 122
167  71 186
122 186  71
186 122 167

Original data:

Data                                  Coding
D0  (up): c0 c9 fc 5f 6e b1 dd 15     C0  (down): 00 00 00 00 00 00 00 00
D1  (up): 91 67 6b 6f 0a e8 55 0c     C1  (down): 00 00 00 00 00 00 00 00
D2  (up): f2 11 95 64 d0 14 e5 49     C2  (down): 00 00 00 00 00 00 00 00
                                      C3  (down): 00 00 00 00 00 00 00 00
Encoding complete:

Data                                  Coding
D0  (up): c0 c9 fc 5f 6e b1 dd 15     C0  (up): 90 45 c6 d3 44 ae 35 d3
D1  (up): 91 67 6b 6f 0a e8 55 0c     C1  (up): 8e 38 4c 6c 52 a2 23 02
D2  (up): f2 11 95 64 d0 14 e5 49     C2  (up): 18 90 5d 4a 02 25 73 42
                                      C3  (up): df 93 97 a7 17 7c 44 41

Erased 4 random devices:

Data                                    Coding
D0  (down): 00 00 00 00 00 00 00 00     C0  (down): 00 00 00 00 00 00 00 00
D1  (up  ): 91 67 6b 6f 0a e8 55 0c     C1  (up  ): 8e 38 4c 6c 52 a2 23 02
D2  (up  ): f2 11 95 64 d0 14 e5 49     C2  (down): 00 00 00 00 00 00 00 00
                                        C3  (down): 00 00 00 00 00 00 00 00

State of the system after decoding:

Data                                  Coding
D0  (up): c0 c9 fc 5f 6e b1 dd 15     C0  (up): 90 45 c6 d3 44 ae 35 d3
D1  (up): 91 67 6b 6f 0a e8 55 0c     C1  (up): 8e 38 4c 6c 52 a2 23 02
D2  (up): f2 11 95 64 d0 14 e5 49     C2  (up): 18 90 5d 4a 02 25 73 42
```

```
                                    C3  (up): df 93 97 a7 17 7c 44 41

Erased the first 4 devices

Data                                  Coding
D0  (down): 00 00 00 00 00 00 00 00   C0  (down): 00 00 00 00 00 00 00 00
D1  (down): 00 00 00 00 00 00 00 00   C1  (up  ): 8e 38 4c 6c 52 a2 23 02
D2  (down): 00 00 00 00 00 00 00 00   C2  (up  ): 18 90 5d 4a 02 25 73 42
                                      C3  (up  ): df 93 97 a7 17 7c 44 41

Here is the decoding matrix:

130  25 182
252 221  25
108 252 130

And dm_ids:

 4 5 6

After calling Dotprod, we calculate the value of device #0 to be:

D0 : c0 c9 fc 5f 6e b1 dd 15

UNIX>
```

Referring back to the conceptual model in Figure 3, it should be clear in this encoding how the first w bits of C_0 are calculated from the first w bits of each data device:

$$\text{byte 0 of } C_0 = (71 \times \text{ byte 0 of } D_0) \oplus (167 \times \text{ byte 0 of } D_1) \oplus (122 \times \text{ byte 0 of } D_2)$$

where multiplication is in $GF(2^8)$.

However, keep in mind that the implementation actually performs dot products on groups of bytes at a time. So in this example, where each device holds 8 bytes, the dot product is actually:

$$8 \text{ bytes of } C_0 = (71 \times 8 \text{ bytes of } D_0) \oplus (167 \times 8 \text{ bytes of } D_1) \oplus (122 \times 8 \text{ bytes of } D_2)$$

This is accomplished using **galois_w08_region_multiply()**.
Here is a similar example, this time with $w = 16$ and each device holding 16 bytes:

```
UNIX> reed_sol_crs_01 3 4 16 16
The Coding Matrix (the last m rows of the Distribution Matrix):

52231 20482 30723
20482 52231 27502
30723 27502 52231
27502 30723 20482

Original data:

Data                                                  Coding
D0  (up): c0c9 fc5f 6eb1 dd15 9167 6b6f 0ae8 550c     C0  (down): 0000 0000 0000 0000 0000 0000 0000 0000
D1  (up): f211 9564 d014 e549 aebb 332f 69d1 9958     C1  (down): 0000 0000 0000 0000 0000 0000 0000 0000
D2  (up): b446 5f5f ba16 dc6f 3bf7 462d 4808 1839     C2  (down): 0000 0000 0000 0000 0000 0000 0000 0000
```

```
                                                    C3  (down): 0000 0000 0000 0000 0000 0000 0000 0000

Encoding complete:

Data                                                Coding
D0  (up): c0c9 fc5f 6eb1 dd15 9167 6b6f 0ae8 550c   C0  (up): 8b79 e6cb ab06 4bcc b303 e10b 068b 1576
D1  (up): f211 9564 d014 e549 aebb 332f 69d1 9958   C1  (up): 7c5b 6194 7851 e1f5 9db9 340c 6181 a121
D2  (up): b446 5f5f ba16 dc6f 3bf7 462d 4808 1839   C2  (up): 41e2 4d35 a59b 8c16 0608 a65b d8d3 dcf1
                                                    C3  (up): d106 470e d782 63f9 f8e2 1b1e 47de 019c

Erased 4 random devices:

Data                                                  Coding
D0  (up  ): c0c9 fc5f 6eb1 dd15 9167 6b6f 0ae8 550c   C0  (down): 0000 0000 0000 0000 0000 0000 0000 0000
D1  (down): 0000 0000 0000 0000 0000 0000 0000 0000   C1  (down): 0000 0000 0000 0000 0000 0000 0000 0000
D2  (down): 0000 0000 0000 0000 0000 0000 0000 0000   C2  (up  ): 41e2 4d35 a59b 8c16 0608 a65b d8d3 dcf1
                                                      C3  (up  ): d106 470e d782 63f9 f8e2 1b1e 47de 019c

State of the system after decoding:

Data                                                Coding
D0  (up): c0c9 fc5f 6eb1 dd15 9167 6b6f 0ae8 550c   C0  (up): 8b79 e6cb ab06 4bcc b303 e10b 068b 1576
D1  (up): f211 9564 d014 e549 aebb 332f 69d1 9958   C1  (up): 7c5b 6194 7851 e1f5 9db9 340c 6181 a121
D2  (up): b446 5f5f ba16 dc6f 3bf7 462d 4808 1839   C2  (up): 41e2 4d35 a59b 8c16 0608 a65b d8d3 dcf1
                                                    C3  (up): d106 470e d782 63f9 f8e2 1b1e 47de 019c

Erased the first 4 devices

Data                                                  Coding
D0  (down): 0000 0000 0000 0000 0000 0000 0000 0000   C0  (down): 0000 0000 0000 0000 0000 0000 0000 0000
D1  (down): 0000 0000 0000 0000 0000 0000 0000 0000   C1  (up  ): 7c5b 6194 7851 e1f5 9db9 340c 6181 a121
D2  (down): 0000 0000 0000 0000 0000 0000 0000 0000   C2  (up  ): 41e2 4d35 a59b 8c16 0608 a65b d8d3 dcf1
                                                      C3  (up  ): d106 470e d782 63f9 f8e2 1b1e 47de 019c

Here is the decoding matrix:

  130   260   427
  252   448   260
  108   252   130

And dm_ids:

 4 5 6

After calling JER_Slices::Dotprod(), we calculate the value of device #0 to be:

D0 : c0c9 fc5f 6eb1 dd15 9167 6b6f 0ae8 550c

UNIX>
```

In this encoding, the 8 16-bit half-words of C_0 are calculated as:

$$(52231 \times 8 \text{ half-words of } D_0) \oplus (20482 \times 8 \text{ half-words of } D_1) \oplus (30723 \times 8 \text{ half-words of } D_2)$$

using **galois_w16_region_multiply()**.

This program demonstrates usage of **CRS_Generator()**, **JER_Matrix::Print()**, **JER_Slices::Encode()**,

JER_Slices::Decode(), **JER_Slices::Add_Drive_Failure()**, **JER_Slices::Make_Decoding_Matrix()**, and **JER_Slices::Dotprod()**.

- **reed_sol_crs_02.cpp**: This takes four parameters: k, m, w and *packetsize*, and performs a similar example to **reed_sol_crs_01**, except it converts the Cauchy coding matrix to a bit-matrix. $k + m$ must be less than or equal to 2^w and *packetsize* must be a multiple of **8**. It sets up each device to hold a total of $w * packetsize$ bytes. Here, packets are numbered p_0 through p_{w-1} for each device. It then performs the same encoding and decoding as the previous example but with the corresponding bit-matrix procedures.

  Here is a run with 3 data devices and 4 coding devices with $w = 3$ and a *packetsize* of 8 bytes. (Each device will hold $3 * 8 = 24$ bytes.)

```
UNIX> reed_sol_crs_02 3 4 3 8
Last (m * w) rows of the Binary Distribution Matrix:

111 001 101
100 101 111
110 010 011

001 111 010
101 100 011
010 110 101

101 010 111
111 011 100
011 101 110

010 101 001
011 111 101
101 011 010

Original data:

Data                              Coding
D0  (up): p0 :15ddb16e5ffcc9c0    C0  (down): p0 :0000000000000000
          p1 :0c55e80a6f6b6791                 p1 :0000000000000000
          p2 :49e514d0649511f2                 p2 :0000000000000000
D1  (up): p0 :5899d1692f33bbae    C1  (down): p0 :0000000000000000
          p1 :6fdc16ba5f5f46b4                 p1 :0000000000000000
          p2 :391808482d46f73b                 p2 :0000000000000000
D2  (up): p0 :5dc3340b214ef45c    C2  (down): p0 :0000000000000000
          p1 :327837ea636dda66                 p1 :0000000000000000
          p2 :6c2a1c8349b36d81                 p2 :0000000000000000
                                  C3  (down): p0 :0000000000000000
                                              p1 :0000000000000000
                                              p2 :0000000000000000

Encoding complete:

Data                              Coding
D0  (up): p0 :15ddb16e5ffcc9c0    C0  (up): p0 :589c6d7411b9d145
          p1 :0c55e80a6f6b6791              p1 :77cd772d5619c6ee
          p2 :49e514d0649511f2              p2 :280664b745165f02
D1  (up): p0 :5899d1692f33bbae    C1  (up): p0 :75c0eca15ad2c1b5
          p1 :6fdc16ba5f5f46b4              p1 :5af35fbe3e84d47b
```

```
          p2 :391808482d46f73b              p2 :0af9075177fa0356
D2  (up): p0 :5dc3340b214ef45c    C2  (up): p0 :3075ac666fa6dd3d
          p1 :327837ea636dda66              p1 :5b6a674d0755fa70
          p2 :6c2a1c8349b36d81              p2 :4b8a261a4ba814cc
                                    C3  (up): p0 :01fe2da824ad4685
                                              p1 :7a041bc93e29e59f
                                              p2 :38848ca62a1db3db

Erased 4 random devices:

Data                                Coding
D0  (down): p0 :0000000000000000    C0  (down): p0 :0000000000000000
            p1 :0000000000000000                p1 :0000000000000000
            p2 :0000000000000000                p2 :0000000000000000
D1  (up  ): p0 :5899d1692f33bbae    C1  (up  ): p0 :75c0eca15ad2c1b5
            p1 :6fdc16ba5f5f46b4                p1 :5af35fbe3e84d47b
            p2 :391808482d46f73b                p2 :0af9075177fa0356
D2  (down): p0 :0000000000000000    C2  (up  ): p0 :3075ac666fa6dd3d
            p1 :0000000000000000                p1 :5b6a674d0755fa70
            p2 :0000000000000000                p2 :4b8a261a4ba814cc
                                    C3  (down): p0 :0000000000000000
                                                p1 :0000000000000000
                                                p2 :0000000000000000

State of the system after decoding:

Data                              Coding
D0  (up): p0 :15ddb16e5ffcc9c0    C0  (up): p0 :589c6d7411b9d145
          p1 :0c55e80a6f6b6791              p1 :77cd772d5619c6ee
          p2 :49e514d0649511f2              p2 :280664b745165f02
D1  (up): p0 :5899d1692f33bbae    C1  (up): p0 :75c0eca15ad2c1b5
          p1 :6fdc16ba5f5f46b4              p1 :5af35fbe3e84d47b
          p2 :391808482d46f73b              p2 :0af9075177fa0356
D2  (up): p0 :5dc3340b214ef45c    C2  (up): p0 :3075ac666fa6dd3d
          p1 :327837ea636dda66              p1 :5b6a674d0755fa70
          p2 :6c2a1c8349b36d81              p2 :4b8a261a4ba814cc
                                    C3  (up): p0 :01fe2da824ad4685
                                              p1 :7a041bc93e29e59f
                                              p2 :38848ca62a1db3db

Erased the first 4 devices

Data                                Coding
D0  (down): p0 :0000000000000000    C0  (down): p0 :0000000000000000
            p1 :0000000000000000                p1 :0000000000000000
            p2 :0000000000000000                p2 :0000000000000000
D1  (down): p0 :0000000000000000    C1  (up  ): p0 :75c0eca15ad2c1b5
            p1 :0000000000000000                p1 :5af35fbe3e84d47b
            p2 :0000000000000000                p2 :0af9075177fa0356
D2  (down): p0 :0000000000000000    C2  (up  ): p0 :3075ac666fa6dd3d
            p1 :0000000000000000                p1 :5b6a674d0755fa70
            p2 :0000000000000000                p2 :4b8a261a4ba814cc
                                    C3  (up  ): p0 :01fe2da824ad4685
                                                p1 :7a041bc93e29e59f
                                                p2 :38848ca62a1db3db

Here is the decoding matrix:
```

```
101 011 010
111 110 011
011 111 101

001 011 011
101 110 110
010 111 111

110 001 101
001 101 111
100 010 011

And dm_ids:

 4 5 6

After calling JER_Slices::Dotprod(), we calculate the value of device #0 to be:

D0 :
p0: 15ddb16e5ffcc9c0
p1: 0c55e80a6f6b6791
p2: 49e514d0649511f2

UNIX>
```

In this encoding, the first packet of C_0 is computed according to the six ones in the first row of the coding matrix:

$$C_0p_0 = D_0p_0 \oplus D_0p_1 \oplus D_0p_2 \oplus D_1p_2 \oplus D_2p_0 \oplus D_2p_2$$

These dot-products are accomplished with **galois_region_xor()** .

This program demonstrates usage of **CRS_Generator()**, **CRS_Convert_To_Bitmatrix()**, **JER_Matrix::Print()**, **JER_Slices::Encode()**, **JER_Slices::Decode()**, **JER_Slices::Add_Drive_Failure()**, **JER_Slices::Make_Decoding_Matrix()**, and **JER_Slices::Dotprod()**.

- **reed_sol_crs_03.cpp**: This takes three parameters: k, m and w. It performs the same coding/decoding as in **reed_sol_crs_02**, except it uses bit-matrix scheduling instead of bit-matrix operations. The *packetsize* is set at **8** bytes. It creates a "dumb" and "smart" schedule for encoding, encodes with them and prints out how many XORs each took. The smart schedule will outperform the dumb one.

  Finally, it erases m random devices and decodes with **JER_Slices::Decode_Schedule_Lazy()**. It decodes once with a smart schedule and once with a dumb schedule.

  Example:

```
UNIX> reed_sol_crs_03 3 4 3
Last (m * w) rows of the Binary Distribution Matrix:

111 001 101
100 101 111
110 010 011

001 111 010
101 100 011
010 110 101
```

```
101 010 111
111 011 100
011 101 110

010 101 001
011 111 101
101 011 010

Original data:

Data                                Coding
D0  (up): p0 :0000000000204a16    C0  (down): p0 :0000000000000000
          p1 :05542a27169c39e2                 p1 :0000000000000000
          p2 :2eab49560bb0b8b4                 p2 :0000000000000000
D1  (up): p0 :0bd068c93e5d224d    C1  (down): p0 :0000000000000000
          p1 :436c8d9c3a2adfda                 p1 :0000000000000000
          p2 :1dd8ca796a67c4f1                 p2 :0000000000000000
D2  (up): p0 :7742fa2948b62d69    C2  (down): p0 :0000000000000000
          p1 :472e195d0681a900                 p1 :0000000000000000
          p2 :622ebb7e026bccdf                 p2 :0000000000000000
                                    C3  (down): p0 :0000000000000000
                                                p1 :0000000000000000
                                                p2 :0000000000000000

Dumb Schedule Encoding Complete: - 432 XOR'd bytes

Data                                Coding
D0  (up): p0 :0000000000204a16    C0  (up): p0 :234be85f3db6ee07
          p1 :05542a27169c39e2              p1 :444afaba1846e41c
          p2 :2eab49560bb0b8b4              p2 :63380598287cc9f1
D1  (up): p0 :0bd068c93e5d224d    C1  (up): p0 :3ce17f27632128d2
          p1 :436c8d9c3a2adfda              p1 :007b83bc3127b530
          p2 :1dd8ca796a67c4f1              p2 :58848e25583625c3
D2  (up): p0 :7742fa2948b62d69    C2  (up): p0 :3f859cc07de665ce
          p1 :472e195d0681a900              p1 :0209debd05f7fd02
          p2 :622ebb7e026bccdf              p2 :0d9b22b50721e383
                                    C3  (up): p0 :717233e940cd1381
                                              p1 :6bf70d0a39e15986
                                              p2 :373117ee5d5c4089

Smart Schedule Encoding Complete: - 264 XOR'd bytes

Data                                Coding
D0  (up): p0 :0000000000204a16    C0  (up): p0 :234be85f3db6ee07
          p1 :05542a27169c39e2              p1 :444afaba1846e41c
          p2 :2eab49560bb0b8b4              p2 :63380598287cc9f1
D1  (up): p0 :0bd068c93e5d224d    C1  (up): p0 :3ce17f27632128d2
          p1 :436c8d9c3a2adfda              p1 :007b83bc3127b530
          p2 :1dd8ca796a67c4f1              p2 :58848e25583625c3
D2  (up): p0 :7742fa2948b62d69    C2  (up): p0 :3f859cc07de665ce
          p1 :472e195d0681a900              p1 :0209debd05f7fd02
          p2 :622ebb7e026bccdf              p2 :0d9b22b50721e383
                                    C3  (up): p0 :717233e940cd1381
                                              p1 :6bf70d0a39e15986
                                              p2 :373117ee5d5c4089
```

```
Erased 4 random devices:

Data                                Coding
D0  (up  ): p0 :0000000000204a16    C0  (up  ): p0 :234be85f3db6ee07
            p1 :05542a27169c39e2                p1 :444afaba1846e41c
            p2 :2eab49560bb0b8b4                p2 :63380598287cc9f1
D1  (down): p0 :0000000000000000    C1  (down): p0 :0000000000000000
            p1 :0000000000000000                p1 :0000000000000000
            p2 :0000000000000000                p2 :0000000000000000
D2  (down): p0 :0000000000000000    C2  (down): p0 :0000000000000000
            p1 :0000000000000000                p1 :0000000000000000
            p2 :0000000000000000                p2 :0000000000000000
                                    C3  (up  ): p0 :717233e940cd1381
                                                p1 :6bf70d0a39e15986
                                                p2 :373117ee5d5c4089

Decoded using a smart schedule - 272 XOR'd bytes

Data                              Coding
D0  (up): p0 :0000000000204a16    C0  (up): p0 :234be85f3db6ee07
          p1 :05542a27169c39e2              p1 :444afaba1846e41c
          p2 :2eab49560bb0b8b4              p2 :63380598287cc9f1
D1  (up): p0 :0bd068c93e5d224d    C1  (up): p0 :3ce17f27632128d2
          p1 :436c8d9c3a2adfda              p1 :007b83bc3127b530
          p2 :1dd8ca796a67c4f1              p2 :58848e25583625c3
D2  (up): p0 :7742fa2948b62d69    C2  (up): p0 :3f859cc07de665ce
          p1 :472e195d0681a900              p1 :0209debd05f7fd02
          p2 :622ebb7e026bccdf              p2 :0d9b22b50721e383
                                  C3  (up): p0 :717233e940cd1381
                                            p1 :6bf70d0a39e15986
                                            p2 :373117ee5d5c4089

Erased the same 4 devices:

Data                                Coding
D0  (up  ): p0 :0000000000204a16    C0  (up  ): p0 :234be85f3db6ee07
            p1 :05542a27169c39e2                p1 :444afaba1846e41c
            p2 :2eab49560bb0b8b4                p2 :63380598287cc9f1
D1  (down): p0 :0000000000000000    C1  (down): p0 :0000000000000000
            p1 :0000000000000000                p1 :0000000000000000
            p2 :0000000000000000                p2 :0000000000000000
D2  (down): p0 :0000000000000000    C2  (down): p0 :0000000000000000
            p1 :0000000000000000                p1 :0000000000000000
            p2 :0000000000000000                p2 :0000000000000000
                                    C3  (up  ): p0 :717233e940cd1381
                                                p1 :6bf70d0a39e15986
                                                p2 :373117ee5d5c4089

Decoded using a dumb schedule - 352 XOR'd bytes

Data                              Coding
D0  (up): p0 :0000000000204a16    C0  (up): p0 :234be85f3db6ee07
          p1 :05542a27169c39e2              p1 :444afaba1846e41c
          p2 :2eab49560bb0b8b4              p2 :63380598287cc9f1
D1  (up): p0 :0bd068c93e5d224d    C1  (up): p0 :3ce17f27632128d2
          p1 :436c8d9c3a2adfda              p1 :007b83bc3127b530
          p2 :1dd8ca796a67c4f1              p2 :58848e25583625c3
```

```
D2  (up): p0 :7742fa2948b62d69    C2  (up): p0 :3f859cc07de665ce
          p1 :472e195d0681a900              p1 :0209debd05f7fd02
          p2 :622ebb7e026bccdf              p2 :0d9b22b50721e383
                                  C3  (up): p0 :717233e940cd1381
                                            p1 :6bf70d0a39e15986
                                            p2 :373117ee5d5c4089

UNIX>
```

This demonstrates usage of **CRS_Generator()**, **CRS_Convert_To_Bitmatrix()**, **JER_Matrix::Print()**, **JER_Slices::Create_Encode_Schedule()**, **JER_Slices::Encode()**, **JER_Slices::Add_Drive_Failure()**, and **JER_Slices::Decode_Schedule_Lazy()**.

- **reed_sol_crs_04.cpp**: This takes two parameters: n and w. It calls **CRS_N_Ones()** to determine the number of ones in the bit-matrix representation of n in $GF(2^w)$. Then it converts n to a bit-matrix, prints it and confirms the number of ones:

```
UNIX> reed_sol_crs_04 1 5
# Ones: 5

Bitmatrix has 5 ones

10000
01000
00100
00010
00001

UNIX> reed_sol_crs_04 31 5
# Ones: 16

Bitmatrix has 16 ones

11110
11111
10001
11000
11100
UNIX>
```

This demonstrates usage of **JER_Matrix::Set()**, **JER_Matrix::Matrix_To_Bitmatrix()**, **JER_Matrix::Get()**, **CRS_N_Ones()**, and **JER_Matrix::Print()**.

- **reed_sol_crs_05.cpp**: This takes three parameters: k, m and w. (In this and the following examples, **PacketSize** = **8**.) It calls **CRS_Generator()** to create an Cauchy matrix, converts it to a bit-matrix then encodes and decodes with it. Smart scheduling is employed. Lastly, it uses **CRS_XY_Generator()** to create the same Cauchy matrix. It verifies that the two matrices are indeed identical. Example:

```
UNIX> reed_sol_crs_05 3 3 3
The generator matrix has 46 ones:
6 7 2
5 2 7
1 3 4
```

```
Original data:

Data                              Coding
D0  (up): p0 :15ddb16e5ffcc9c0    C0  (down): p0 :0000000000000000
          p1 :0c55e80a6f6b6791                 p1 :0000000000000000
          p2 :49e514d0649511f2                 p2 :0000000000000000
D1  (up): p0 :5899d1692f33bbae    C1  (down): p0 :0000000000000000
          p1 :6fdc16ba5f5f46b4                 p1 :0000000000000000
          p2 :391808482d46f73b                 p2 :0000000000000000
D2  (up): p0 :5dc3340b214ef45c    C2  (down): p0 :0000000000000000
          p1 :327837ea636dda66                 p1 :0000000000000000
          p2 :6c2a1c8349b36d81                 p2 :0000000000000000

Smart Schedule Encoding Complete: - 224 XOR'd bytes

Data                              Coding
D0  (up): p0 :15ddb16e5ffcc9c0    C0  (up): p0 :27c72fc21f6711c3
          p1 :0c55e80a6f6b6791               p1 :70f8a08577598c22
          p2 :49e514d0649511f2               p2 :5550bd8d470398df
D1  (up): p0 :5899d1692f33bbae    C1  (up): p0 :23014e4e16411ad1
          p1 :6fdc16ba5f5f46b4               p1 :75a7f9fa47aea93b
          p2 :391808482d46f73b               p2 :15baa4354280a14e
D2  (up): p0 :5dc3340b214ef45c    C2  (up): p0 :46245fa53ee45f33
          p1 :327837ea636dda66               p1 :5c5a0cf8189fda57
          p2 :6c2a1c8349b36d81               p2 :2ec822aa7e7139a0

Erased 3 random devices:

Data                                Coding
D0  (up  ): p0 :15ddb16e5ffcc9c0    C0  (up  ): p0 :27c72fc21f6711c3
            p1 :0c55e80a6f6b6791                 p1 :70f8a08577598c22
            p2 :49e514d0649511f2                 p2 :5550bd8d470398df
D1  (up  ): p0 :5899d1692f33bbae    C1  (down): p0 :0000000000000000
            p1 :6fdc16ba5f5f46b4                 p1 :0000000000000000
            p2 :391808482d46f73b                 p2 :0000000000000000
D2  (down): p0 :0000000000000000    C2  (down): p0 :0000000000000000
            p1 :0000000000000000                 p1 :0000000000000000
            p2 :0000000000000000                 p2 :0000000000000000

Decode (lazily) with a smart schedule: 192 XOR'd bytes

Data                              Coding
D0  (up): p0 :15ddb16e5ffcc9c0    C0  (up): p0 :27c72fc21f6711c3
          p1 :0c55e80a6f6b6791               p1 :70f8a08577598c22
          p2 :49e514d0649511f2               p2 :5550bd8d470398df
D1  (up): p0 :5899d1692f33bbae    C1  (up): p0 :23014e4e16411ad1
          p1 :6fdc16ba5f5f46b4               p1 :75a7f9fa47aea93b
          p2 :391808482d46f73b               p2 :15baa4354280a14e
D2  (up): p0 :5dc3340b214ef45c    C2  (up): p0 :46245fa53ee45f33
          p1 :327837ea636dda66               p1 :5c5a0cf8189fda57
          p2 :6c2a1c8349b36d81               p2 :2ec822aa7e7139a0

Generated the identical matrix using CRS_XY_Generator()

UNIX>
```

This demonstrates usage of **CRS_Generator()**, **CRS_N_Ones()**, **JER_Matrix::Print()**, **JER_Gen_T::Genmatrix_To_Genbitmatrix()**, **JER_Gen_T()::Create_Encode_Schedule()**, **JER_Slices::Encode()**, **JER_Slices::Decode_Schedule_Lazy()**, **CRS_XY_Generator()**, and **JER_Matrix::Get()**.

- **reed_sol_crs_06.cpp**: This example differs from **reed_sol_crs_05.cpp** in two ways. First, **reed_sol_crs_06.cpp** improves the matrix with **CRS_Improve_Generator()**. Secondly, this does not use **CRS_XY_Generator()** to check that the matrices are identical.
  Example:

```
UNIX> reed_sol_crs_06 3 3 3 | head -n 10
The generator matrix has 46 ones:
6 7 2
5 2 7
1 3 4

The improved generator matrix has 34 ones:
1 1 1
5 1 2
1 4 7

UNIX>
```

This demonstrates usage of **CRS_Generator()**, **CRS_N_Ones()**, **JER_Matrix::Get()**, **JER_Matrix::Print()**, **CRS_Improve_Generator()**, **JER_Gen_T::Genmatrix_To_Genbitmatrix()**, **JER_Gen_T()::Create_Encode_Schedule()**, **JER_Slices::Encode()**, and **JER_Slices::Decode_Schedule_Lazy()**.

- **reed_sol_crs_07.cpp**: This is identical to the previous two, except it calls **CRS_Good_Generator_Bitmatrix()**. Note, when $m = 2$, $w \leq 11$ and $k \leq 1023$, these are optimal Cauchy encoding matrices. That's not to say that they are optimal RAID-6 matrices (RDP encoding [CEG+04], and Liberation encoding [Pla08b] achieve this), but they are the best Cauchy matrices.

```
UNIX> reed_sol_crs_07 10 2 8 | head -n 4
Matrix has 229 ones

  1   1   1   1   1   1   1   1   1   1
  1   2 142   4  71   8  70 173   3  35

UNIX>

UNIX> reed_sol_crs_06 10 2 8 | head -n 8
The generator matrix has 608 ones:
142 244  71 167 122 186 173 157 221 152
244 142 167  71 186 122 157 173 152 221

The improved generator matrix has 354 ones:
  1   1   1   1   1   1   1   1   1   1
 82 200 151 172   1 225 166 158  44  13

UNIX>

UNIX> reed_sol_crs_05 10 2 8 | head -n 4
```

```
Matrix has 608 ones

142 244  71 167 122 186 173 157 221 152
244 142 167  71 186 122 157 173 152 221

UNIX>
```

This demonstrates usage of **CRS_Good_Generator_Bitmatrix()**, **JER_Gen_T()::Bitmatrix_To_Matrix()**, **JER_Matrix::Get()**, **JER_Matrix::Print()**, **JER_Gen_T()::Create_Encode_Schedule()**, **JER_Slices::Encode()**, and **JER_Slices::Decode_Schedule_Lazy()**.

9 Part 4 of the Library: Bitmatrix-based Coding (bitmatrices.h)

The **bitmatrices.h** file contains code to create three types of generator matrices: generalized EVENODD, generalized RDP, and mimal density RAID-6. All of these generators have **PDrive** = *true* and **Systematic** = *true*.

Minimal Density RAID-6 codes are MDS codes based on binary matrices which satisfy a lower-bound on the number of non-zero entries. Unlike Cauchy coding, the bit-matrix elements do not correspond to elements in $GF(2^w)$. Instead, the bit-matrix itself has the proper MDS property. Minimal Density RAID-6 codes perform faster than Reed-Solomon and Cauchy Reed-Solomon codes for the same parameters. Liberation coding, Liber8tion coding, and Blaum-Roth coding are three examples of this kind of coding that are supported in Jerasure. Note that since these codes are RAID-6, m must be 2.

Each coding method places restrictions on the values of k, m, and w. The rules are listed below:

- generalized EVENODD - k $\leq$ w+1, m $\in$ (2,4,5,6,7,8), w+1 is prime. Additional rules apply when m$\neq$2, and **Gen_Evenodd_Smallest_W()** should be used to select a valid value of w. [BBV]
- generalized RDP - k $\leq$ w, m $\geq$ 2, w+1 is prime
- Blaum-Roth - k $\leq$ w, m = 2, w+1 is prime [BR99]
- Liber8tion - k $\leq$ w, m = 2, w = 8 [Pla08a]
- Liberation - k $\leq$ w, m = 2, w is prime [Pla08b]

9.1 EVENODD, RDP, and minimal density RAID-6 generators - bitmatrices.cpp

The functions found in bitmatrices.cpp are as follows:

- **int R6_Min_Density_Smallest_W(int k)**: Given **k**, this returns the smallest w that is a valid value for a minimal density RAID 6 code (Liberation, Blaum-Roth, or Liber8tion). This may return a value >32, which cannot be used to encode or decode in Jerasure. If no valid w exists, -1 is returned.
- **int Gen_Evenodd_Smallest_W(int k, int m)**: Given **k** and **m**, this returns the smallest legal w for a generalized EVENODD algorithm. For m=2, this returns the smallest $w\geq k$-1 where w+1 is prime. For $m\in$(4,5,6,7,8), this returns the smallest valid value of w listed in Table I of [BBV]. This may return a value >32, which cannot be used to encode or decode in Jerasure. If no valid w exists, -1 is returned.

- **JER_Gen_T *R6_Min_Density_Generator(int k, int w)**: This function creates and returns a **JER_Gen_T** object containing a minimal density matrix. The type of minimal density coding matrix, dependent upon **w**, is either Blaum-Roth, Liberation, or Liber8tion. Returns NULL on failure.
- **int R6_Min_Density_Generator(int k, int w, JER_Gen_T &g)**: This modifies an existing **JER_Gen_T** object, **g**, to contain a minimal density matrix. A new **JER_Matrix** object is created if the generator's matrix is NULL. Otherwise, the existing matrix is resized and transformed appropriately. Any schedules in the generator's **Schedules** map are deleted with a call to **Delete_Schedules()**. Returns 0 on success, -1 on failure.
- **JER_Gen_T *Gen_Evenodd_Generator(int k, int m, int w)**: Creates and returns a generator object containing a generalized EVENODD matrix. Returns NULL on failure.
- **int Gen_Evenodd_Generator(int k, int m, int w, JER_Gen_T &g)**: Modifies **g** to be a generator object containing a generalized EVENODD matrix. Returns 0 on success, -1 on failure.
- **JER_Gen_T *Gen_RDP_Generator(int k, int m, int w)**: Creates and returns a generator object containing a generalized RDP matrix. Returns NULL on failure.
- **int Gen_RDP_Generator(int k, int m, int w, JER_Gen_T &g)**: Modifies **g** to be a generator object containing a generalized RDP matrix. Returns 0 on success, -1 on failure.

9.2 Example programs

- **bitmatrices_evenodd_01.cpp**: This takes two parameters: k and m. M must be one of the following: 2,4,5,6,7,8. This uses **Gen_Evenodd_Smallest_W()** to determine the smallest valid value of w that can be used. As in other examples, **PacketSize** is 8. It sets up an Evenodd bit-matrix and uses it for encoding and decoding. It then encodes w*8 bytes by converting the bit-matrix to a dumb schedule. The dumb schedule is used because that schedule cannot be improved upon. For decoding, smart scheduling is used as it gives a big savings over dumb scheduling.

```
UNIX> ./bitmatrices_evenodd_01 5 4
Smallest w is 4

Coding Bit-Matrix:

1000 1000 1000 1000 1000
0100 0100 0100 0100 0100
0010 0010 0010 0010 0010
0001 0001 0001 0001 0001

1000 0001 0011 0110 1100
0100 1001 0010 0101 1010
0010 0101 1010 0100 1001
0001 0011 0110 1100 1000

1000 0011 1100 0001 0110
0100 0010 1010 1001 0101
0010 1010 1001 0101 0100
0001 0110 1000 0011 1100

1000 0110 0001 1100 0011
```

```
0100 0101 1001 1010 0010
0010 0100 0101 1001 1010
0001 1100 0011 1000 0110

Original data:

Data                              Coding
D0  (up): p0 :0000000000204a16    C0  (down): p0 :0000000000000000
          p1 :05542a27169c39e2                p1 :0000000000000000
          p2 :2eab49560bb0b8b4                p2 :0000000000000000
          p3 :0bd068c93e5d224d                p3 :0000000000000000
D1  (up): p0 :436c8d9c3a2adfda    C1  (down): p0 :0000000000000000
          p1 :1dd8ca796a67c4f1                p1 :0000000000000000
          p2 :7742fa2948b62d69                p2 :0000000000000000
          p3 :472e195d0681a900                p3 :0000000000000000
D2  (up): p0 :622ebb7e026bccdf    C2  (down): p0 :0000000000000000
          p1 :204d531e262b5331                p1 :0000000000000000
          p2 :7020241c440a0e2a                p2 :0000000000000000
          p3 :75cb1c4c41f9a5e1                p3 :0000000000000000
D3  (up): p0 :67bc26ff181e279e    C3  (down): p0 :0000000000000000
          p1 :7172c06f4909e153                p1 :0000000000000000
          p2 :09d3bfba685ae502                p2 :0000000000000000
          p3 :7e10e8ab0f262618                p3 :0000000000000000
D4  (up): p0 :726b82306469f5c5
          p1 :0cead30e206c7b8a
          p2 :028a2bec306eac4e
          p3 :56e5747a57287c72

Dumb Schedule Encoding Complete: - 800 XOR'd bytes

Data                              Coding
D0  (up): p0 :0000000000204a16    C0  (up): p0 :3495922d44168b48
          p1 :05542a27169c39e2              p1 :4559a02133b534fb
          p2 :2eab49560bb0b8b4              p2 :229003355f38d2bb
          p3 :0bd068c93e5d224d              p3 :11c0f109212b74c6
D1  (up): p0 :436c8d9c3a2adfda    C1  (up): p0 :44e50fe66604c2c3
          p1 :1dd8ca796a67c4f1              p1 :0eb51be27c15dfd2
          p2 :7742fa2948b62d69              p2 :33af33355b7f7f54
          p3 :472e195d0681a900              p3 :0f74981f2735c837
D2  (up): p0 :622ebb7e026bccdf    C2  (up): p0 :027f1b5d7573a04d
          p1 :204d531e262b5331              p1 :23bb264c7837d000
          p2 :7020241c440a0e2a              p2 :0ee8621b5cfd9ff8
          p3 :75cb1c4c41f9a5e1              p3 :0a26e5c83d9e4a5f
D3  (up): p0 :67bc26ff181e279e    C3  (up): p0 :5df0951a5559109e
          p1 :7172c06f4909e153              p1 :24a2ec9879c253ff
          p2 :09d3bfba685ae502              p2 :0fb8abf5453ad298
          p3 :7e10e8ab0f262618              p3 :3953c96163ff62f7
D4  (up): p0 :726b82306469f5c5
          p1 :0cead30e206c7b8a
          p2 :028a2bec306eac4e
          p3 :56e5747a57287c72

Erased 4 random devices:

Data                                Coding
D0  (up  ): p0 :0000000000204a16    C0  (down): p0 :0000000000000000
             p1 :05542a27169c39e2                p1 :0000000000000000
```

```
            p2 :2eab49560bb0b8b4                p2 :0000000000000000
            p3 :0bd068c93e5d224d                p3 :0000000000000000
D1  (down): p0 :0000000000000000    C1  (down): p0 :0000000000000000
            p1 :0000000000000000                p1 :0000000000000000
            p2 :0000000000000000                p2 :0000000000000000
            p3 :0000000000000000                p3 :0000000000000000
D2  (up  ): p0 :622ebb7e026bccdf    C2  (up  ): p0 :027f1b5d7573a04d
            p1 :204d531e262b5331                p1 :23bb264c7837d000
            p2 :7020241c440a0e2a                p2 :0ee8621b5cfd9ff8
            p3 :75cb1c4c41f9a5e1                p3 :0a26e5c83d9e4a5f
D3  (up  ): p0 :67bc26ff181e279e    C3  (up  ): p0 :5df0951a5559109e
            p1 :7172c06f4909e153                p1 :24a2ec9879c253ff
            p2 :09d3bfba685ae502                p2 :0fb8abf5453ad298
            p3 :7e10e8ab0f262618                p3 :3953c96163ff62f7
D4  (down): p0 :0000000000000000
            p1 :0000000000000000
            p2 :0000000000000000
            p3 :0000000000000000

Decode (lazily) with a smart schedule: 880 XOR'd bytes

Data                            Coding
D0  (up): p0 :0000000000204a16    C0  (up): p0 :3495922d44168b48
          p1 :05542a27169c39e2              p1 :4559a02133b534fb
          p2 :2eab49560bb0b8b4              p2 :229003355f38d2bb
          p3 :0bd068c93e5d224d              p3 :11c0f109212b74c6
D1  (up): p0 :436c8d9c3a2adfda    C1  (up): p0 :44e50fe66604c2c3
          p1 :1dd8ca796a67c4f1              p1 :0eb51be27c15dfd2
          p2 :7742fa2948b62d69              p2 :33af33355b7f7f54
          p3 :472e195d0681a900              p3 :0f74981f2735c837
D2  (up): p0 :622ebb7e026bccdf    C2  (up): p0 :027f1b5d7573a04d
          p1 :204d531e262b5331              p1 :23bb264c7837d000
          p2 :7020241c440a0e2a              p2 :0ee8621b5cfd9ff8
          p3 :75cb1c4c41f9a5e1              p3 :0a26e5c83d9e4a5f
D3  (up): p0 :67bc26ff181e279e    C3  (up): p0 :5df0951a5559109e
          p1 :7172c06f4909e153              p1 :24a2ec9879c253ff
          p2 :09d3bfba685ae502              p2 :0fb8abf5453ad298
          p3 :7e10e8ab0f262618              p3 :3953c96163ff62f7
D4  (up): p0 :726b82306469f5c5
          p1 :0cead30e206c7b8a
          p2 :028a2bec306eac4e
          p3 :56e5747a57287c72

UNIX>
```

This demonstrates usage of **Gen_Evenodd_Smallest_W()**, **Gen_Evenodd_Generator()**, **JER-_Matrix::Print()**, **JER_Gen_T::Create_Encode_Schedule()**, **JER_Slices::Encode()**, and **JER-_Slices::Decode_Schedule_Lazy()**.

- **bitmatrices_rdp_01.cpp**: This takes three parameters: k, m, and w. K must be less than or equal to w. $W+1$ must be prime. As in other examples, **PacketSize** is 8. It sets up an RDP bitmatrix and converts it to a dumb schedule. The schedule is used to encode w*8 bytes. A dumb schedule is used because it cannot be improved upon. For decoding, smart scheduling is used as it gives a big savings over dumb scheduling.

```
UNIX> ./bitmatrices_rdp_01 4 3 4
```

```
Coding Bit-Matrix:

1000 1000 1000 1000
0100 0100 0100 0100
0010 0010 0010 0010
0001 0001 0001 0001

1100 0100 0101 0110
0110 1010 0010 0011
0011 0101 1001 0001
0001 0010 0100 1000

1010 0011 0110 0010
0101 0001 0011 1001
0010 1000 0001 0100
1001 1100 1000 1010


Original data:

Data                                Coding
D0  (up): p0 :0000000000204a16    C0  (down): p0 :0000000000000000
          p1 :05542a27169c39e2                p1 :0000000000000000
          p2 :2eab49560bb0b8b4                p2 :0000000000000000
          p3 :0bd068c93e5d224d                p3 :0000000000000000
D1  (up): p0 :436c8d9c3a2adfda    C1  (down): p0 :0000000000000000
          p1 :1dd8ca796a67c4f1                p1 :0000000000000000
          p2 :7742fa2948b62d69                p2 :0000000000000000
          p3 :472e195d0681a900                p3 :0000000000000000
D2  (up): p0 :622ebb7e026bccdf    C2  (down): p0 :0000000000000000
          p1 :204d531e262b5331                p1 :0000000000000000
          p2 :7020241c440a0e2a                p2 :0000000000000000
          p3 :75cb1c4c41f9a5e1                p3 :0000000000000000
D3  (up): p0 :67bc26ff181e279e
          p1 :7172c06f4909e153
          p2 :09d3bfba685ae502
          p3 :7e10e8ab0f262618

Dumb Schedule Encoding Complete: - 432 XOR'd bytes

Data                                Coding
D0  (up): p0 :0000000000204a16    C0  (up): p0 :46fe101d207f7e8d
          p1 :05542a27169c39e2              p1 :49b3732f13d94f71
          p2 :2eab49560bb0b8b4              p2 :201a28d96f567ef5
          p3 :0bd068c93e5d224d              p3 :47258573760308b4
D1  (up): p0 :436c8d9c3a2adfda    C1  (up): p0 :35abd0d93a5a4584
          p1 :1dd8ca796a67c4f1              p1 :183267c94cc6bed5
          p2 :7742fa2948b62d69              p2 :1678bd2215bfb82e
          p3 :472e195d0681a900              p3 :3b63e70148de7b8b
D2  (up): p0 :622ebb7e026bccdf    C2  (up): p0 :4779629a4fdcced2
          p1 :204d531e262b5331              p1 :55edadb73c8b18e2
          p2 :7020241c440a0e2a              p2 :697e18e9396a23dc
          p3 :75cb1c4c41f9a5e1              p3 :59250d171c1f7d33
D3  (up): p0 :67bc26ff181e279e
          p1 :7172c06f4909e153
          p2 :09d3bfba685ae502
          p3 :7e10e8ab0f262618
```

```
Erased 3 random devices:

Data                                Coding
D0  (down): p0 :0000000000000000    C0  (up): p0 :46fe101d207f7e8d
            p1 :0000000000000000              p1 :49b3732f13d94f71
            p2 :0000000000000000              p2 :201a28d96f567ef5
            p3 :0000000000000000              p3 :47258573760308b4
D1  (down): p0 :0000000000000000    C1  (up): p0 :35abd0d93a5a4584
            p1 :0000000000000000              p1 :183267c94cc6bed5
            p2 :0000000000000000              p2 :1678bd2215bfb82e
            p3 :0000000000000000              p3 :3b63e70148de7b8b
D2  (up  ): p0 :622ebb7e026bccdf    C2  (up): p0 :4779629a4fdcced2
            p1 :204d531e262b5331              p1 :55edadb73c8b18e2
            p2 :7020241c440a0e2a              p2 :697e18e9396a23dc
            p3 :75cb1c4c41f9a5e1              p3 :59250d171c1f7d33
D3  (down): p0 :0000000000000000
            p1 :0000000000000000
            p2 :0000000000000000
            p3 :0000000000000000

Decode (lazily) with a smart schedule: 504 XOR'd bytes

Data                              Coding
D0  (up): p0 :0000000000204a16    C0  (up): p0 :46fe101d207f7e8d
          p1 :05542a27169c39e2              p1 :49b3732f13d94f71
          p2 :2eab49560bb0b8b4              p2 :201a28d96f567ef5
          p3 :0bd068c93e5d224d              p3 :47258573760308b4
D1  (up): p0 :436c8d9c3a2adfda    C1  (up): p0 :35abd0d93a5a4584
          p1 :1dd8ca796a67c4f1              p1 :183267c94cc6bed5
          p2 :7742fa2948b62d69              p2 :1678bd2215bfb82e
          p3 :472e195d0681a900              p3 :3b63e70148de7b8b
D2  (up): p0 :622ebb7e026bccdf    C2  (up): p0 :4779629a4fdcced2
          p1 :204d531e262b5331              p1 :55edadb73c8b18e2
          p2 :7020241c440a0e2a              p2 :697e18e9396a23dc
          p3 :75cb1c4c41f9a5e1              p3 :59250d171c1f7d33
D3  (up): p0 :67bc26ff181e279e
          p1 :7172c06f4909e153
          p2 :09d3bfba685ae502
          p3 :7e10e8ab0f262618

UNIX>
```

This demonstrates usage of **Gen_RDP_Generator()**, **JER_Matrix::Print()**, **JER_Gen_T::Create_Encode_Schedule()**, **JER_Slices::Encode()**, and **JER_Slices::Decode_Schedule_Lazy()**.

- **bitmatrices_min_den_r6_01.cpp**: This takes one parameter (k). **R6_Min_Density_Smallest_W()** is used to determine the smallest value of W that can be use with the given k. As in other examples, **PacketSize** is 8, and w*8 bytes are encoded and decoded. A minimal density bitmatrix is created. The specific type of coding matrix depends on w. If w+1 is prime and k≤w, a Blaum-Roth coding matrix is used. If w=8 and k≤w, a Liber8tion coding matrix is used. If w is prime and k≤w, a Liberation coding matrix is used. This example encodes by converting the bitmatrix to a dumb schedule. The dumb schedule is used because that schedule cannot be improved upon. For decoding, smart scheduling is used as it gives a big savings over dumb scheduling.

```
UNIX> ./bitmatrices_min_den_r6_01 3
When K is 3, the smallest valid value of W is 3.

Liberation coding

Coding Bit-Matrix:

100 100 100
010 010 010
001 001 001

100 010 001
010 011 100
001 100 110


Original data:

Data                              Coding
D0  (up): p0 :0000000000204a16    C0  (down): p0 :0000000000000000
          p1 :05542a27169c39e2                 p1 :0000000000000000
          p2 :2eab49560bb0b8b4                 p2 :0000000000000000
D1  (up): p0 :0bd068c93e5d224d    C1  (down): p0 :0000000000000000
          p1 :436c8d9c3a2adfda                 p1 :0000000000000000
          p2 :1dd8ca796a67c4f1                 p2 :0000000000000000
D2  (up): p0 :7742fa2948b62d69
          p1 :472e195d0681a900
          p2 :622ebb7e026bccdf

Dumb Schedule Encoding Complete: - 112 XOR'd bytes

Data                              Coding
D0  (up): p0 :0000000000204a16    C0  (up): p0 :7c9292e076cb4532
          p1 :05542a27169c39e2              p1 :0116bee62a374f38
          p2 :2eab49560bb0b8b4              p2 :515d385163bcb09a
D1  (up): p0 :0bd068c93e5d224d    C1  (up): p0 :214236e238615913
          p1 :436c8d9c3a2adfda              p1 :2ca297eb0e670fa0
          p2 :1dd8ca796a67c4f1              p2 :1517c2eb7bda1e90
D2  (up): p0 :7742fa2948b62d69
          p1 :472e195d0681a900
          p2 :622ebb7e026bccdf

Erased 2 random devices:

Data                                Coding
D0  (up  ): p0 :0000000000204a16    C0  (down): p0 :0000000000000000
            p1 :05542a27169c39e2                p1 :0000000000000000
            p2 :2eab49560bb0b8b4                p2 :0000000000000000
D1  (up  ): p0 :0bd068c93e5d224d    C1  (up  ): p0 :214236e238615913
            p1 :436c8d9c3a2adfda                p1 :2ca297eb0e670fa0
            p2 :1dd8ca796a67c4f1                p2 :1517c2eb7bda1e90
D2  (down): p0 :0000000000000000
            p1 :0000000000000000
            p2 :0000000000000000

Decode (lazily) with a smart schedule: 112 XOR'd bytes
```

```
Data                              Coding
D0  (up): p0 :0000000000204a16    C0  (up): p0 :7c9292e076cb4532
          p1 :05542a27169c39e2              p1 :0116bee62a374f38
          p2 :2eab49560bb0b8b4              p2 :515d385163bcb09a
D1  (up): p0 :0bd068c93e5d224d    C1  (up): p0 :214236e238615913
          p1 :436c8d9c3a2adfda              p1 :2ca297eb0e670fa0
          p2 :1dd8ca796a67c4f1              p2 :1517c2eb7bda1e90
D2  (up): p0 :7742fa2948b62d69
          p1 :472e195d0681a900
          p2 :622ebb7e026bccdf

UNIX>
```

This demonstrates usage of **R6_Min_Density_Smallest_W**, **R6_Min_Density_Generator()**, **JER-_Matrix::Print()**, **JER_Gen_T::Create_Encode_Schedule()**, **JER_Slices::Encode()**, and **JER-_Slices::Decode_Schedule_Lazy()**.

10 Example Application 1: Encoder and Decoder

The programs **encoder** and **decoder** are used to encode and decode a single file. When the file is encoded, it is split into **N** new files. **K** of these new files represent data slices, and **M** represent coding slices. The **decoder** can reconstruct the original file if **K** or more of the new files are not deleted.

10.1 Encoder - encoder.cpp

This program is used to encode a file using any of the available methods in **Jerasure**. It takes seven parameters:

- **inputfile** or negative number S: either the file to be encoded or a negative number S indicating that a random file of size $-S$ should be used rather than an existing file
- **K**: number of data files
- **M**: number of coding files
- **coding_technique**: must be one of the following:
  - *no_coding*
  - *reed_sol_van*: calls **RS_Classic_Generator()**
  - *reed_sol_r6_op*: calls **RS_R6_Generator()**
  - *cauchy_orig*: calls **CRS_Generator()**, **CRS_Convert_To_Bitmatrix()**, and **JER_Gen_T::Create_-Encode_Schedule(smart = *true*)**
  - *cauchy_good*: calls **CRS_Good_Generator_Bitmatrix()** and **JER_Gen_T::Create_Encode_-Schedule(smart = *true*)**
  - *r6_min_density*: calls **R6_Min_Density_Generator()** and **JER_Gen_T::Create_Encode_-Schedule(smart = *true*)**
  - *gen_rdp*: calls **Gen_RDP_Generator()**

  - *gen_evenodd*: calls **Gen_Evenodd_Generator()**

- **W**: word size
- (**packetsize**): For non-bitmatrix based encoding methods, this argument is ignored and set equal to the size of one slice. Therefore, for non-bitmatrix based coding methods, **PacketsPerSlice** is one. For bitmatrix-based coding methods, this must be a multiple of 8.
- (**buffersize**): This is the approximate size of data (in bytes) to be read in at a time. It is adjusted to be a multiple of (**K** * **W** * **packetsize**). **Buffersize** is an optional parameter. If it is not given, or it is set to 0, the entire file is read in at once.

Encoder reads in **inputfile** (or creates random data), splits the file into **K** blocks, and encodes the file into **M** blocks. It also creates a file containing meta-data to be used for decoding purposes. It writes all of these files into a directory named *Coding*. The output of this program is the rate of encoding and the total rate of the program; both are given in MB/sec.

```
UNIX> ls -l Movie.wmv
-rwxr-xr-x   1 plank  plank  55211097 Aug 14 10:52 Movie.wmv
UNIX> encoder Movie.wmv 6 2 r6_min_density 7 1024 500000
Encoding (MB/sec): 1405.3442614500
En_Total (MB/sec): 5.8234765527
UNIX> ls -l Coding
total 143816
-rw-r--r--   1 plank  plank  9203712 Aug 14 10:54 Movie_k1.wmv
-rw-r--r--   1 plank  plank  9203712 Aug 14 10:54 Movie_k2.wmv
-rw-r--r--   1 plank  plank  9203712 Aug 14 10:54 Movie_k3.wmv
-rw-r--r--   1 plank  plank  9203712 Aug 14 10:54 Movie_k4.wmv
-rw-r--r--   1 plank  plank  9203712 Aug 14 10:54 Movie_k5.wmv
-rw-r--r--   1 plank  plank  9203712 Aug 14 10:54 Movie_k6.wmv
-rw-r--r--   1 plank  plank  9203712 Aug 14 10:54 Movie_m1.wmv
-rw-r--r--   1 plank  plank  9203712 Aug 14 10:54 Movie_m2.wmv
-rw-r--r--   1 plank  plank       54 Aug 14 10:54 Movie_meta.txt
UNIX> echo "" | awk '{ print 9203712*6 }'
55222272
UNIX>
```

In the above example a 52.7 MB movie file is broken into six data and two coding blocks using Liberation codes with **W**= 7 and **packetsize** = 1K. A buffer of 500000 bytes is specified but **encoder** modifies the buffersize so that it is a multiple of (**K** * **W** * **packetsize**).

The new directory, **Coding**, contains the six files **Movie_k1.wmv** through **Movie_k6.wmv** (which are parts of the original file) plus the two encoded files **Movie_m1.wmv** and **Movie_m2.wmv**. Note that the file sizes are multiples of 7 and 1024 as well – the original file was padded with zeros so that it would encode properly. The metadata file, **Movie_meta.txt** contains all information relevant to **decoder**.

10.2 Decoder - decoder.cpp

This program is used in conjunction with **encoder** to decode any files remaining after erasures and reconstruct the original file. The only parameter for **decoder** is **inputfile**, the original file that was encoded. This file does not have to exist; the file name is needed only to find files created by **encoder**, which should be in the *Coding* directory.

After some number of erasures, the program locates the surviving files from **encoder** and recreates the original file if at least **K** of the files still exist. The rate of decoding and the total rate of running the program are given as output.

Continuing the previous example, suppose that Movie_k2.wmv and Movie_m1.wmv are erased.

```
UNIX> rm Coding/Movie_k1.wmv Coding/Movie_k2.wmv
UNIX> mv Movie.wmv Old-Movie.wmv
UNIX> decoder Movie.wmv
Decoding (MB/sec): 1167.8230894030
De_Total (MB/sec): 16.0071713224

UNIX> ls -l Coding
total 215704
-rw-r--r--   1 plank  plank  55211097 Aug 14 11:02 Movie_decoded.wmv
-rw-r--r--   1 plank  plank   9203712 Aug 14 10:54 Movie_k3.wmv
-rw-r--r--   1 plank  plank   9203712 Aug 14 10:54 Movie_k4.wmv
-rw-r--r--   1 plank  plank   9203712 Aug 14 10:54 Movie_k5.wmv
-rw-r--r--   1 plank  plank   9203712 Aug 14 10:54 Movie_k6.wmv
-rw-r--r--   1 plank  plank   9203712 Aug 14 10:54 Movie_m1.wmv
-rw-r--r--   1 plank  plank   9203712 Aug 14 10:54 Movie_m2.wmv
-rw-r--r--   1 plank  plank        54 Aug 14 10:54 Movie_meta.txt
UNIX> diff Coding/Movie_decoded.wmv Old-Movie.wmv
UNIX>
```

This reads in all of the remaining files and creates **Movie_decoded.wmv** which, as shown by the **diff** command, is identical to the original **Movie.wmv**. Note that **decoder** does not recreate the lost data files – just the original.

10.3 Judicious selection of buffer and packet sizes

In our tests, the buffer and packet sizes have as much impact on performance as the code used. Initial performance results are in [SP08]; however these will be fleshed out more thoroughly. To give a compelling example, look at the following coding times for a randomly created 256M file on a MacBook Pro (2.16 GHz processor, 32KB L1 cache, 2MB L2 cache):

```
UNIX> encoder -268435456 6 2 r6_min_density 7 1024 50000000
Encoding (MB/sec): 172.6522847357
En_Total (MB/sec): 141.8509585895
UNIX> encoder -268435456 6 2 r6_min_density 7 1024 5000000
Encoding (MB/sec): 1066.8065148470
En_Total (MB/sec): 526.1761192874
UNIX> encoder -268435456 6 2 r6_min_density 7 10240 5000000
Encoding (MB/sec): 1084.6304288755
En_Total (MB/sec): 555.2568545979
UNIX> encoder -268435456 6 2 r6_min_density 7 102400 5000000
Encoding (MB/sec): 943.4553565388
En_Total (MB/sec): 525.2790538399
UNIX>
```

When using these routines, one should pay attention to packet and buffer sizes.

11 Example Application 2: Personal File Archiving

The personal file archiving example is composed of two separate programs. First, the **personal_archiving** program takes a folder of files as input, and creates a coding file, **pc**. The original folder of files and the coding file can then be stored for later use. The second program, **personal_retrieval** restores the original files by fixing data corruptions. This is only possible when **M**, or less, blocks have failed. The following sections detail the usage and behavior of these two programs.

11.1 Personal archiving - personal_archiving.cpp

This program is used to create an archive of files. It takes the following arguments:

- **M**: number of coding blocks to create
- **W**: word size for classic Reed Solomon coding
- **blocksize**: size of each block in bytes
- **dir**: name of directory containing files to archive
- **pc**: name of the output file containing coding data

Personal_archiving creates a temporary tar file of the directory **dir**. The tar file contains the original file data and meta-data such as file modification times. All of the data in the tar file is split into blocks of size blocksize. For each block of file data, the program encodes **M** coding blocks. The program holds (**M**+1) blocks of size **blocksize** in memory. All encoding is performed with a classic Reed Solomon generator matrix. The coding data is then written to the file **pc**. **Pc** also contains a checksum for each block. This checksum is used by the **personal_retrieval** program to determine which blocks are corrupt.

For this demonstration, a folder containing 3 files of various sizes is archived.

```
UNIX> ls -lh my_files/
total 6.0M
-rwxrwxrwx 1 root root 1.0M 2011-07-31 15:08 rand_1MB
-rwxrwxrwx 1 root root 2.0M 2011-07-31 15:09 rand_2MB
-rwxrwxrwx 1 root root 3.0M 2011-07-31 15:09 rand_3MB
UNIX> ./personal_archiving

usage: ./personal_archiving m w blocksize dir pc

UNIX> ./personal_archiving 10 8 1048576 my_files pc
Tar'ing the input folder

Listing tar file's contents (note: All files are zero-padded to a multiple of 512 bytes.
Each file also contains 512 extra bytes of meta-data.)
     my_files/            Original size: 0          Size in .tar: 512
     my_files/rand_1MB    Original size: 1048576    Size in .tar: 1049088
     my_files/rand_2MB    Original size: 2097152    Size in .tar: 2097664
     my_files/rand_3MB    Original size: 3145728    Size in .tar: 3146240

Total size of the .tar file: 6293504 bytes
Given a blocksize of 1048576 bytes, k=10

Creating generator matrix...
```

```
Encoding...
     File 1/4: my_files
     File 2/4: my_files/rand_1MB
     File 3/4: my_files/rand_2MB
     File 4/4: my_files/rand_3MB

Archiving complete

UNIX> ls -l pc
-rwxrwxrwx 1 root root 10485968 2011-07-31 15:13 pc
```

Given a blocksize of 1MB, the tar file of the my_files folder was split into 10 blocks ($\mathbf{K} = 10$). As the blocksize is increased, the number of data blocks, **K**, decreases.

The resulting **pc** file contains meta-data such as **M** and **W**, 10 coding blocks, and 20 checksums. The original my_files folder and contents were not modified.

11.2 Personal retrieval - personal_retrieval.cpp

Personal_retrieval accepts two arguments:

- **dir**: name of directory containing files previously archived
- **pc**: name of the input file containing coding data

Meta-data from **pc** is read to determine the arguments used to archive **dir**. **Personal_retrieval** creates a temporary tar file of the **dir** folder. This tar file is split into **K** blocks, and **M** blocks of coding data are read from **pc**. For each data and coding block read in, a checksum is computed. These new checksums are compared to those stored in **pc**. This process determines which blocks are corrupt. If **M** or fewer blocks are corrupt, the failed data blocks are decoded.

A maximum of ($\mathbf{M} + 1$) blocks of size **blocksize** are held in memory during decoding. Please note that **personal_retrieval** does not fix corrupt coding blocks. Additionally, blocks are marked corrupt if the headers in the temporary archiving and retrieval tar files differ. Therefore, decoding occurs if a file's size or modification time changes, because this information is stored in the tar's header. Finally, **personal_retrieval** does not tolerate deleted files.

The archived state of the my_files directory is shown below:

```
UNIX> ls -lh my_files/
total 6.0M
-rwxrwxrwx 1 root root 1.0M 2011-07-31 15:08 rand_1MB
-rwxrwxrwx 1 root root 2.0M 2011-07-31 15:09 rand_2MB
-rwxrwxrwx 1 root root 3.0M 2011-07-31 15:09 rand_3MB
ls -lh | grep my_files
drwxrwxrwx 1 root root    0 2011-07-31 15:09 my_files
```

Personal_retrieval will not perform decoding if all of the blocks are unchanged.

```
UNIX> ./personal_retrieval my_files pc

Retrieved coding information:
     k=10
     m=10
```

```
     w=8
     method=reed-sol
     blocksize=1048576

Tar'ing the input folder

Listing tar file's contents (note: All files are zero-padded to a multiple of 512 bytes.
Each file also contains 512 extra bytes of meta-data.)
     my_files/            Original size: 0          Size in .tar: 512
     my_files/rand_1MB    Original size: 1048576    Size in .tar: 1049088
     my_files/rand_2MB    Original size: 2097152    Size in .tar: 2097664
     my_files/rand_3MB    Original size: 3145728    Size in .tar: 3146240

Creating the generator matrix

Comparing data checksums
Comparing coding checksums

Number of corrupt data blocks: 0
Number of corrupt coding blocks: 0

No corrupt blocks to decode
```

A data block failure is generated by touching a single file in my_files. A block fails because a file's modification time changes. When **personal_retrieval** creates the tar file of my_files, the header data for one of the files will create a new, unmatched, checksum.

```
UNIX> touch my_files/rand_1MB
UNIX> ls -lh my_files/
total 6.0M
-rwxrwxrwx 1 root root 1.0M 2011-07-31 16:13 rand_1MB
-rwxrwxrwx 1 root root 2.0M 2011-07-31 15:09 rand_2MB
-rwxrwxrwx 1 root root 3.0M 2011-07-31 15:09 rand_3MB
UNIX> ./personal_retrieval my_files/ pc

Retrieved coding information:
     k=10
     m=10
     w=8
     method=reed-sol
     blocksize=1048576

Tar'ing the input folder

Listing tar file's contents (note: All fi
 to a multiple of 512 bytes. Each file al
ra bytes of meta-data.)
     my_files/            Original size:
.tar: 512
     my_files/rand_1MB    Original size:
.tar: 1049088
     my_files/rand_2MB    Original size:
.tar: 2097664
     my_files/rand_3MB    Original size:
.tar: 3146240

Creating generator matrix
```

```
Comparing data checksums
Comparing coding checksums

Number of corrupt data blocks: 1
Number of corrupt coding blocks: 0

my_files/
my_files/rand_1MB
my_files/rand_2MB
my_files/rand_3MB
UNIX> ls -lh my_files/
total 6.0M
-rwxrwxrwx 1 root root 1.0M 2011-07-31 15:08 rand_1MB
-rwxrwxrwx 1 root root 2.0M 2011-07-31 15:09 rand_2MB
-rwxrwxrwx 1 root root 3.0M 2011-07-31 15:09 rand_3MB
```

After decoding, the rand_1MB's modification time is restored to its archived state.

12 Example Application 3: RAID

The **raid** program is used to demonstrate how Jerasure may be used in order to manage a RAID system. All encoding and decoding uses a classic Reed Solomon generator matrix. The program reads **K** files of identical size and writes the encoding of the original data to **N** files. Next, **M** partial failures and **M** - 2 complete disk failures are created. Finally, the program attempts to recover from these. Because, this program generates 2 * (**M** - 1) failures, there exist scenarios in which some data loss occurs. However, in most cases, this program demonstrates recovery from both partial and disk failures, even when more than **M** failures are present.

12.1 raid.cpp

This example program requires five parameters. The following lists and describes these parameters:

- **N**: total number of data and coding files the program will handle. **N** lets you indirectly specify the number of coding files, **M**, that will be created (**M** = **N** - **K**).
- **K**: number of data files existing in the **raid_dir** folder
- **W**: word size
- **packetsize**: Each slice is setup to hold **W** packets of **packetsize** bytes.
- **raid_dir**: The name of an existing directory. This directory should contain **K** files. All files should be the same size, and named file0, file1, ... file(**K**-1).

The following demonstrates usage of the **raid** program:
First, a folder for holding the data files is created.

```
UNIX> mkdir raid_test
```

Five files containing 1 MB of random data are created inside the new folder. Please note that the naming convention of file0, file1, ... file(**K**-1) must be followed.

```
UNIX> dd if=/dev/urandom of=./raid_test/file0 bs=1M count=1
UNIX> dd if=/dev/urandom of=./raid_test/file1 bs=1M count=1
UNIX> dd if=/dev/urandom of=./raid_test/file2 bs=1M count=1
UNIX> dd if=/dev/urandom of=./raid_test/file3 bs=1M count=1
UNIX> dd if=/dev/urandom of=./raid_test/file4 bs=1M count=1
UNIX> ls -lh raid_test
-rwxrwxrwx 1 root root 1.0M 2011-07-31 12:42 file0
-rwxrwxrwx 1 root root 1.0M 2011-07-31 12:42 file1
-rwxrwxrwx 1 root root 1.0M 2011-07-31 12:42 file2
-rwxrwxrwx 1 root root 1.0M 2011-07-31 12:42 file3
-rwxrwxrwx 1 root root 1.0M 2011-07-31 12:42 file4
```

For this example, a copy of the original files is made. The original files will be used to confirm that decoding worked.

```
UNIX> cp -r raid_test raid_test_orig
```

Next, the raid simulation is run.

```
UNIX> ./raid
usage: ./raid n k w packetsize raid_dir
UNIX> ./raid 10 5 8 1024 raid_test
Drive files opened.
Performing encoding.
Encoding successful.
Generating 5 partial failures.
Drive 4 failed at 838080 for 38986 bytes
Drive 1 failed at 333008 for 181714 bytes
Drive 5 failed at 244654 for 340766 bytes
Drive 0 failed at 526408 for 393259 bytes
Drive 1 failed at 980060 for 18706 bytes
Failing drive 3.
Failing drive 4.
Failing drive 0.
Performing partial failure recovery and decoding.
Files successfully decoded.
```

After the simulation runs, the raid directory contains a total of **N** files. **M** new files contain the coding data.

```
UNIX> ls -lh raid_test
total 10M
-rwxrwxrwx 1 root root 1.0M 2011-07-31 12:52 file0
-rwxrwxrwx 1 root root 1.0M 2011-07-31 12:52 file1
-rwxrwxrwx 1 root root 1.0M 2011-07-31 12:52 file2
-rwxrwxrwx 1 root root 1.0M 2011-07-31 12:52 file3
-rwxrwxrwx 1 root root 1.0M 2011-07-31 12:52 file4
-rwxrwxrwx 1 root root 1.0M 2011-07-31 12:52 file5
-rwxrwxrwx 1 root root 1.0M 2011-07-31 12:52 file6
-rwxrwxrwx 1 root root 1.0M 2011-07-31 12:52 file7
-rwxrwxrwx 1 root root 1.0M 2011-07-31 12:52 file8
-rwxrwxrwx 1 root root 1.0M 2011-07-31 12:52 file9
```

If decoding corrected the failures, the first **K** files should be the original data. For this demonstration, the diff command is used to confirm that the original data was recovered.

```
UNIX> diff raid_test/file0 raid_test_orig/file0
UNIX> diff raid_test/file1 raid_test_orig/file1
UNIX> diff raid_test/file2 raid_test_orig/file2
UNIX> diff raid_test/file3 raid_test_orig/file3
UNIX> diff raid_test/file4 raid_test_orig/file4
```

References

[Anv07] H. P. Anvin. The mathematics of RAID-6. `http://kernel.org/pub/linux/kernel/people/hpa/raid6.pdf`, 2007.

[BBBM95] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Transactions on Computing*, 44(2):192– 202, February 1995.

[BBV] M. Blaum, J. Bruck, and A. Vardy. MDS array codes with independent parity symbols. *IEEE Transactions on Information Theory*, 42(2):529– 542, March 1996.

[BKK+95] J. Blomer, M. Kalfane, M. Karpinski, R. Karp, M. Luby, and D. Zuckerman. An XOR-based erasure-resilient coding scheme. Technical Report TR-95-048, International Computer Science Institute, August 1995.

[BR99] M. Blaum and R. M. Roth. On lowest density MDS codes. *IEEE Transactions on Information Theory*, 45(1):46–59, January 1999.

[CEG+04] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row diagonal parity for double disk failure correction. In *4th Usenix Conference on File and Storage Technologies*, San Francisco, CA, March 2004.

[FDBS05a] G. Feng, R. Deng, F. Bao, and J. Shen. New efficient MDS array codes for RAID Part I: Reed-Solomon-like codes for tolerating three disk failures. *IEEE Transactions on Computers*, 54(9):1071–1080, September 2005.

[FDBS05b] G. Feng, R. Deng, F. Bao, and J. Shen. New efficient MDS array codes for RAID Part II: Rabin-like codes for tolerating multiple (≥ 4) disk failures. *IEEE Transactions on Computers*, 54(12):1473–1483, Decemeber 2005.

[HDRT05] J. L. Hafner, V. Deenadhayalan, K. K. Rao, and A. Tomlin. Matrix methods for lost data reconstruction in erasure codes. In *FAST-2005: 4th Usenix Conference on File and Storage Technologies*, pages 183–196, San Francisco, December 2005.

[HX05] C. Huang and L. Xu. STAR: An efficient coding scheme for correcting triple storage node failures. In *FAST-2005: 4th Usenix Conference on File and Storage Technologies*, pages 197–210, San Francisco, December 2005.

[PD05] J. S. Plank and Y. Ding. Note: Correction to the 1997 tutorial on Reed-Solomon coding. *Software – Practice & Experience*, 35(2):189–194, February 2005.

[Pla97] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience*, 27(9):995–1012, September 1997.

[Pla07] J. S. Plank. Fast Galois Field arithmetic library in C/C++. Technical Report CS-07-593, University of Tennessee, April 2007.

[Pla08a] J. S. Plank. A new minimum density RAID-6 code with a word size of eight. In *NCA-08: 7th IEEE International Symposium on Network Computing Applications*, Cambridge, MA, July 2008.

[Pla08b] J. S. Plank. The RAID-6 Liberation codes. In *FAST-2008: 6th Usenix Conference on File and Storage Technologies*, pages 97–110, San Jose, February 2008.

[Pre89] F. P. Preparata. Holographic dispersal and recovery of information. *IEEE Transactions on Information Theory*, 35(5):1123–1124, September 1989.

[PX06] J. S. Plank and L. Xu. Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications. In *NCA-06: 5th IEEE International Symposium on Network Computing Applications*, Cambridge, MA, July 2006.

[Rab89] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the Association for Computing Machinery*, 36(2):335–348, April 1989.

[RS60] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8:300–304, 1960.

[SP08] C. D. Schuman and J. S. Plank. A performance comparison of open-source erasure coding libraries for storage applications. Technical Report UT-CS-08-625, University of Tennessee, August 2008.