GUAVA

D.Justo Peralta López.
jperalta@ual.es
Dpto. Álgebra y Análisis Matemático.
Universidad de Almería.

12 de enero de 2004

Índice general

Capítulo 1

Códigos con GAP

GUAVA es una librería en donde se implementa algoritmos de teoría de códigos en **GAP**. Estos algoritmos nos permiten crear códigos , manipularlos y obtener sus propiedades más importantes.

GUAVA ha sido diseñado para la construcción y análisis de códigos y sus funciones se pueden dividir en tres clases o categorías:

1. Construcción de códigos: **GUAVA** puede construir códigos sin ningún tipo de restricción, códigos lineales y códigos cíclicos.

2. Manipulaciones de códigos: Transformaciones de un código en otro, o construcción de un código nuevo a partir de otros dos.

3. Cálculo de información de un código: **GUAVA** calcula los parámetros más importante de un código.El resultado es guardado en un registro para evitar volver a calcular ciertos parmetros cuando dicha informacin haga falta.

1.1. Cargando GUAVA

Antes de empezar, **GUAVA** tiene que ser cargado. Para ello escribe:

```
gap> RequirePackage( "guava" );
```

Y en su pantalla aparecerá

```
   ____                              |
  /            \           /   --+--  Version 1.51
 /      |    | |\\        //|    |
|    _  |    | | \\      // |     Jasper Cramwinckel
|     \ |    | |--\\    //--|     Erik Roijackers
 \     ||    | |   \\  //   |     Reinald Baart
  \___/  \___/ |    \\//    |     Eric Minkes
                                   Lea Ruscio
true
```

En el caso de que no exista suficiente memoria, **GUAVA** le avisará.

1.2. Palabra código

Una palabra código es simplemente un vector sobre un cuerpo finito, e internamente es representado mediante un registro. Además, una palabra puede ser definida de diferentes formas e imprimida en diferentes formatos.

```
gap> Codeword( <obj> [, <n>] [, <F>] )
```

Codeword devuelve una palabra código o una lista de palabras construidas a partir de **obj**. El objeto **obj** puede ser un vector, una cadena, un polinomio o una palabra código. También puede ser una lista de los objetos anteriormente citados y no necesariamente del mismo tipo.

Si el número **n** es especificado, todas las palabras tendrán la misma longitud **n**. Los elementos de **obj** que tienen longitud mayor que **n** son reducidas cortando las últimas posiciones, y aquellos que tiene menor longitud que **n** son completados con ceros al final.

Si el Cuerpo de Galois **F** se especifica, todas la palabras son definidas sobre dicho cuerpo. Este es el único camino para asegurar que todos los elementos de **obj** sean del mismo tipo. Además, si hemos especificado **F**, todos los elementos de **obj** deben estar definidos sobre **F**, o como mínimo como enteros. Pasar de un cuerpo a otro no está permitido, y si **F** no se especifica, los vectores o cadenas con enteros se convierte al cuerpo más pequeño posible.

Toda cadena en **obj** debe ser una cadena de números, sin espacios, comas o cualquier otro carácter; y los números deben estar entre 0 y 9. Las cadenas son convertidos a una palabra sobre **F** o, si **F** es omitido, sobre el menor cuerpo de Galois posible. Fijemonos que ya que todos los números son interpretados como de un solo dígito, cuerpo de Galois de tamaño mayor que 10 no son representados correctamente usando cadenas.

Todo polinomio en **obj** es convertida a una palabra de longitud **n**, o si la longitud no se especifica, la longitud de la palabra vendrá dada por el grado del polinomio. Si **F** es especificado, entonces el polinomio en **obj** debe ser definido sobre **F**.

```
gap> c := Codeword([0,1,1,1,0]);
[ 0 1 1 1 0 ]
gap> VectorCodeword( c );
[ 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2) ]
gap> c2 := Codeword([0,1,1,1,0], GF(3));
[ 0 1 1 1 0 ]
gap> VectorCodeword( c2 );
[ 0*Z(3), Z(3)^0, Z(3)^0, Z(3)^0, 0*Z(3) ]
gap> Codeword([c, c2, "0110"]);
[ [ 0 1 1 1 0 ], [ 0 1 1 1 0 ], [ 0 1 1 0 ] ]
gap> p := UnivariatePolynomial(GF(2), [Z(2)^0, 0*Z(2), Z(2)^0]);
Z(2)^0+x_1^2
gap> Codeword(p);
x^2 + 1
```

```
Codeword( <obj>, <C> )
```

De esta forma, los elementos de **obj** son convertidos a elementos del mismo espacio vectorial que los elementos del código **C**. Esto es lo mismo que llamar **Codeword** con longitud de palabra la misma que en **C** y el cuerpo usado para **C**.

```
gap> C := WholeSpaceCode(7,GF(5));
a cyclic [7,7,1]0 whole space code over GF(5)
gap> Codeword(["0220110", [1,1,1]], C);
[ [ 0 2 2 0 1 1 0 ], [ 1 1 1 0 0 0 0 ] ]
gap> Codeword(["0220110", [1,1,1]], 7, GF(5));
[ [ 0 2 2 0 1 1 0 ], [ 1 1 1 0 0 0 0 ] ] |
```

```
IsCodeword( <obj> )
```

Esta función devuelve **true** si **obj** es una palabra código, en caso contrario devuelve **false**. La función devuelve un error si **obj** no posee un valor determinado.

```
gap> IsCodeword(1);
false
gap> IsCodeword(ReedMullerCode(2,3));
false
gap> IsCodeword("11111");
false
gap> IsCodeword(Codeword("11111"));
true
```

1.3. Comparación de plabras

```
C1 = C2
C1<> C2
```

El operador = devuelve *true* si c_1 y c_2 son iguales, y *falso* en caso contrario. El operador distinto <> devuelve *true* si c_1 y c_2 son distintos y *false* en caso contrario.

Nótese que dos palabras no pueden ser iguales si son definidas sobre cuerpos diferentes o espacios vectoriales distintos. También se puede comparar palabras código con otros objetos, pero no es recomendable ya que no siempre funciona correctamente.

```
gap> P := UnivariatePolynomial(GF(2), Z(2)*[1,0,0,1]);
Z(2)^0+x_1^3

gap> c := Codeword(P, GF(2));
x^3 + 1
gap> P = c;        # codeword operation
true
gap> c = P;        # polynomial operation
false
gap> c2 := Codeword("1001", GF(2));
[ 1 0 0 1 ]
gap> c = c2;
true
```

1.4. Operaciones con palabras del código

Todas las operaciones se debe realizar con palabras de la misma longitud y definidas sobre el mismo cuerpo finito.

$c_1 + c_2$

El operador '+évalúa la suma de dos palabras c_1 y c_2.

$c_1 - c_2$

El operador '-évalúa la diferencia de dos palabras c_1 y c_2.

$C + c$

$c + C$

En este caso, el operador '+'calcula el coconjunto del código C , resultante de suma la palabra c a todas las palabras del código.

En general, estas operaciones se pueden realizar con vectores, cadenas o polinomios, aunque en algunos casos no es muy recomendable. Normalmente, los vectores, cadenas y polinomios primero son convertidos a palabras códigos, y después las operaciones son realizadas normalmente. Aun así es recomendable que como mínimo uno de los dos sumandos sea una palabra código.

1.5. Palabra código como vectores

```
VectorCodeword( <obj> [, <n>] [, <F>] )
VectorCodeword( <obj>, <C> )
```

VectorCodeword devuelve un vector o una lista de vectores sobre un cuerpo de Galois finito. El objeto **obj** puede ser una cadena, un polinomio, una palabra código o una mezcla de ellos. Como se puede comprobar, el objeto **obj** es tratado de la misma forma que en la función **Codeword**.

```
gap> a := Codeword("011011");; VectorCodeword(a);
[ 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0, Z(2)^0 ]
gap> VectorCodeword( [ 0, 1, 2, 1, 2, 1 ] );
[ 0*Z(3), Z(3)^0, Z(3), Z(3)^0, Z(3), Z(3)^0 ]
gap> VectorCodeword( [ 0, 0, 0, 0], GF(9) );
[ 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ]
```

1.6. Palabras códigos como polinomios y vectores

```
PolyCodeword( <obj> [, <n>] [, <F>] )
PolyCodeword( <obj>, <C> )
```

Funciona de la misma forma que **Codeword**

```
gap> a := Codeword("011011");; PolyCodeword(a);
Z(2)^0*(X(GF(2))^5 + X(GF(2))^4 + X(GF(2))^2 + X(GF(2)))
gap> PolyCodeword( [ 0, 1, 2, 1, 2 ] );
Z(3)^0*(2*X(GF(3))^4 + X(GF(3))^3 + 2*X(GF(3))^2 + X(GF(3)))
gap> PolyCodeword( [ 0, 0, 0, 0], GF(9) );
0*X(GF(3^2))^0
```

```
TreatAsVector( <obj> )
```

Esta función hace que **obj** sea tratado como un vector. Además **obj** puede ser una palabra código o una lista de palabras código.

```
gap> B := BinaryGolayCode();
a cyclic [23,12,7]3 binary Golay code over GF(2)
gap> c := CodewordNr(B, 4);
x^22 + x^20 + x^17 + x^14 + x^13 + x^12 + x^11 + x^10
gap> TreatAsVector(c);
gap> c;
[ 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 1 0 0 1 0 1 ] |
```

De la misma forma, hay funciones que permiten que un objeto sea tratado como un polinomio, **TreatAsPoly**, con la siguiente sintaxis

```
TreatAsPoly( <obj> )
```

```
gap> a := Codeword("00001",GF(2));
[ 0 0 0 0 1 ]
gap> TreatAsPoly(a); a;
x^4
gap> b := NullWord(6,GF(4));
[ 0 0 0 0 0 0 ]
gap> TreatAsPoly(b); b;
0
```

1.7. Palabra nula o vacía

```
NullWord( <n> )
NullWord( <n>, <F> )
NullWord( <C> )
```

La función **NullWord** devuelve una palabra código de longitud **n** sobre un cuerpo **F** formado por ceros. Por defecto, **F** es 'GF(2)'y **n** debe ser mayor que cero. Si sólo se especifica el código **C**, entonces esta función devuelve una palabra nula o vacía con la misma longitud que las palabras del código especificado y sobre el mismo cuerpo definido en **C**.

```
gap> NullWord(8);
[ 0 0 0 0 0 0 0 0 ]
gap> Codeword("0000") = NullWord(4);
true
gap> NullWord(5,GF(16));
[ 0 0 0 0 0 ]
gap> NullWord(ExtendedTernaryGolayCode());
[ 0 0 0 0 0 0 0 0 0 0 0 0 ]
```

1.8. Distancia Hamming

```
DistanceCodeword( <c1>, <c2> )
```

DistanceCodeword devuelve la distancia Hamming entre c_1 y c_2. Ambos argumentos deben ser palabras código con la misma longitud y definidas sobre el mismo cuerpo. La distancia Hamming se define como el número de posiciones diferentes entre dos palabras. Como es lógico, la función en cuestión siempre devuelve un entero mayor que cero y menor que la longitud de las palabras en cuestión.

```
gap> a := Codeword([0, 1, 2, 0, 1, 2]);;
b := NullWord(6, GF(3));;
gap> DistanceCodeword(a, b);
4
gap> DistanceCodeword(b, a);
4
gap> DistanceCodeword(a, a);
0
```

1.9. Soporte y peso de una palabra

```
Support( <c> )
```

Support devuelve las posiciones de una palabra distintas de cero.

```
gap> a := Codeword("012320023002");; Support(a);
[ 2, 3, 4, 5, 8, 9, 12 ]
gap> Support(NullWord(7));
[  ]
```

```
WeightCodeword( <c> )
```

WeightCodeword devuelve el peso Hamming de una palabra, el cual consiste en el número de posiciones distintas de cero de la palabra **c**.

```
gap> WeightCodeword(Codeword("22222"));
5
gap> WeightCodeword(NullWord(3));
0
```

```
gap> C := HammingCode(3);
a linear [7,4,3]1 Hamming (3,2) code over GF(2)
gap> Minimum(List(Elements(C){[2..Size(C)]}, WeightCodeword ) );
3 |
```

1.10. Códigos

Un código es una colección de palabras. Y una palabra es una secuencia de elementos de un cuerpo finito $GF(q)$, donde q es una potencia de un primo. Dependiendo del tipo de código, o más bien de como se ha definido, sus elementos pueden ser vectores o polinomios.

En **GUAVA**, un código puede ser definido especificando todas sus palabras, por medio de su matriz generadora (códigos lineales), utilizando un polinomio generador (códigos cíclicos) o especificando todos sus elementos.

```
gap> C := ElementsCode(["1100", "1010", "0001"], "example code",
>                      GF(2) );
a (4,3,1..4)2..4 example code over GF(2)
```

Un código con parámetros (n, M, d), es un código con palabras de longitud n, tamaño o número de palabras M, y mínima distancia d. Si la mínima distancia no ha sido especificada, ésta es acotada por **GUAVA**

```
a (4,3,1..4)2..4 code over GF(2)
```

En este ejemplo, hemos definido un código binario de longitud 4, tamaño 3 y mínima distancia entre 1 y 4.

```
gap> MinimumDistance(C);
2
gap> C;
a (4,3,2)2..4 example code over GF(2)
```

Si las palabras del código forman un subespacio vectorial de $GF(q)^n$, el código es lineal y se puede definir indicando su base o matriz generadora. También es posible definir este tipo de códigos utilizando la matriz de chequeo o paridad.

```
gap> G := GeneratorMatCode([[1,0,1],[0,1,2]], "demo code", GF(3) );
a linear [3,2,1..2]1 demo code over GF(3)
```

Si el código es lineal y al ciclar una palabra ésta sigue perteneciendo al código, entonces lo llamamos código cíclico. Este tipo de códigos se definen especificando su polinomio generador y todas sus palabras son múltiplos de dicho polinomio módulo $x^n - 1$,donde n es la longitud de palabra del código.

```
gap> G := GeneratorPolCode(Indeterminate(GF(2))+Z(2)^0, 7, GF(2) );
a cyclic [7,6,1..2]1 code defined by generator polynomial over GF(2)
```

Cuando un código es definido por medio de todos sus elementos, es posible que **GUAVA** no distinga si es un código lineal o no.

```
gap> L := Z(2)*[ [0,0,0], [1,0,0], [0,1,1], [1,1,1] ];;
gap> C := ElementsCode( L, GF(2) );
a (3,4,1..3)1 user defined unrestricted code over GF(2)
# so far, {\GUAVA} does not know what kind of code this is
gap> IsLinearCode( C );
true                        # it is linear
gap> C;
a linear [3,2,1]1 user defined unrestricted code over GF(2)
```

1.11. Algunas preguntas sobre códigos

1.11.1. ¿Es un código ?

```
IsCode( <obj> )
```

IsCode devuelve *true* si **obj**, es un código o *false* en caso contrario.

```
gap> IsCode( 1 );
false
gap> IsCode( ReedMullerCode( 2,3 ) );
true
gap> IsCode( This_object_is_unbound );
Error, Variable: 'This_object_is_unbound' must have a value
```

1.11.2. ¿Es lineal?

```
IsLinearCode( <obj> )
```

IsLinearCode chequea si **obj** es un código lineal. Si el código ha sido definido como lineal o cíclico, la función devuelve de forma automática *true*. En caso contrario, ésta función busca una base que genere el código, si lo encuentra devuelve *true* y *false* en el caso contrario.

```
gap> C := ElementsCode( [ [0,0,0],[1,1,1] ], GF(2) );
a (3,2,1..3)1 user defined unrestricted code over GF(2)
gap> IsLinearCode( C );
true
gap> IsLinearCode( ElementsCode( [ [1,1,1] ], GF(2) ) );
false
gap> IsLinearCode( 1 );
false
```

1.11.3. ¿Es cíclico ?

```
IsCyclicCode( <obj> )
```

IsCyclicCode chequea si es cíclico. Si el código ha sido marcado como cíclico, devuelve *true* de forma automática. En caso contrario, busca su polinomio generador. Si lo encuentra devuelve *true* y *false* en caso contrario.

```
gap> C := ElementsCode( [ [0,0,0], [1,1,1] ], GF(2) );
a (3,2,1..3)1 user defined unrestricted code over GF(2)
# {\GUAVA} does not know the code is cyclic
gap> IsCyclicCode( C );      # this command tells {\GUAVA} to find out
true
gap> IsCyclicCode( HammingCode( 4, GF(2) ) );
false
gap> IsCyclicCode( 1 );
false
```

1.11.4. ¿Es perfecto ?

```
IsPerfectCode( <C> )
```

IsPerfectCode devuelve *true* si **C** es perfecto.

```
gap> H := HammingCode(2);
a linear [3,1,3]1 Hamming (2,2) code over GF(2)
gap> IsPerfectCode( H );
true
gap> IsPerfectCode( ElementsCode( [ [1,1,0], [0,0,1] ], GF(2) ) );
true
gap> IsPerfectCode( ReedSolomonCode( 6, 3 ) );
false
gap> IsPerfectCode(BinaryGolayCode());
true
```

1.11.5. MDS Códigos

```
IsMDSCode( <C> )
```

IsMDSCode devuelve *true* si **C** es un *Maximum Distance Seperable* código.Un código lineal con parámetros $[n, k, d]$ es un código MDS si $k = n - d + 1$.

```
gap> C1 := ReedSolomonCode( 6, 3 );
a cyclic [6,4,3]2 Reed-Solomon code over GF(7)
gap> IsMDSCode( C1 );
true    # 6-3+1 = 4
gap> IsMDSCode( QRCode( 23, GF(2) ) );
false
```

1.11.6. Códigos Self Duales

```
IsSelfDualCode( <C> )
```

Un código es *Self Dual* si $C^{\perp} = C$

Si **C** no es lineal, entonces no puede ser self-dual, por lo que devuelve *false*. Un código puede ser self-dual sólo si la dimensión k es igual a la redundancia r.

```
gap> IsSelfDualCode( ExtendedBinaryGolayCode() );
true
gap> C := ReedMullerCode( 1, 3 );
a linear [8,4,4]2 Reed-Muller (1,3) code over GF(2)
gap> DualCode( C ) = C;
true
```

1.11.7. Códigos equivalentes

```
IsEquivalent( <C1>,<C2> )
```

IsEquivalent devuelve *true* si ambos códigos son equivalentes.

```
gap> x:= Indeterminate( GF(2) );; pol:= x^3+x+1;
Z(2)^0+x_1+x_1^3
gap> H := GeneratorPolCode( pol, 7, GF(2));
a cyclic [7,4,1..3]1 code defined by generator polynomial over GF(2)
gap> H = HammingCode(3, GF(2));
false
gap> IsEquivalent(H, HammingCode(3, GF(2)));
true                          # H is equivalent to a Hamming code
gap> CodeIsomorphism(H, HammingCode(3, GF(2)));
(3,4)(5,6,7)
```

La última expresión calcula el isomorfismo entre dos códigos equivalentes.

```
gap> x:= Indeterminate( GF(2) );; pol:= x^3+x+1;
Z(2)^0+x_1+x_1^3
gap> H := GeneratorPolCode( pol, 7, GF(2));
a cyclic [7,4,1..3]1 code defined by generator polynomial over GF(2)
gap> CodeIsomorphism(H, HammingCode(3, GF(2)));
(3,4)(5,6,7)
gap> PermutedCode(H, (3,4)(5,6,7)) = HammingCode(3, GF(2));
true
```

1.11.8. Comparación de códigos

$C_1 = C_2$
$C_1 <> C_2$

Un código es igual a otro si y solo si todos sus elementos son iguales.

```
gap> M := [ [0, 0], [1, 0], [0, 1], [1, 1] ];;
gap> C1 := ElementsCode( M, GF(2) );
a (2,4,1..2)0 user defined unrestricted code over GF(2)
gap> M = C1;
false
gap> C2 := GeneratorMatCode( [ [1, 0], [0, 1] ], GF(2) );
a linear [2,2,1]0 code defined by generator matrix over GF(2)
gap> C1 = C2;
```

```
true
```

De la misma forma se puede determinar si dos códigos son equivalentes usando **IsEquivalent**.

1.12. Operaciones con códigos

$C_1 + C_2$
El operador '+évalúa la suma directa de C_1 y C_2.
$C + c$
$c + C$
En este caso el operador '+évalúa las clase de C después de sumarle a todas sus palabras c
$C_1 * C_2$
El operador * evalúa el producto directo de C_1 y C_2.
El operador **in** nos permite saber si una palabra pertenece al código. **c** puede ser una palabra o lista de palabras. Por supuesto, c y C deben tener la misma longitud de palabras y el mismo cuerpo base.

```
gap> C:= HammingCode( 2 );; eC:= AsSSortedList( C );
[ [ 0 0 0 ], [ 1 1 1 ] ]
gap> [ [ 0, 0, 0, ], [ 1, 1, 1, ] ] in C;
true
gap> [ 0 ] in C;
false
```

$IsSubset(C_1, C_2)$
En este caso, la expresión devuelve *true* si C_1 es un subcódigo de C_2.

```
gap> IsSubset( HammingCode(3), RepetitionCode( 7 ) );
true
gap> IsSubset( RepetitionCode( 7 ), HammingCode( 3 ) );
false
gap> IsSubset( WholeSpaceCode( 7 ), HammingCode( 3 ) );
true
```

1.13. Funciones sobre códigos

En esta sección mostramos una serie de funciones útiles sobre códigos
Función *es finito*

```
gap> IsFinite( RepetitionCode( 1000, GF(11) ) );
true
```

Función *tamaño* de un código

```
Size(<C>)
```

Esta función devuelve el tamaño del código **C**. Si el código es lineal, el tamaño es igual a q^k, donde q es el tamaño del alfabeto o tamaño del cuerpo base de **C** y k es la dimensión de la base.

```
gap> Size( RepetitionCode( 1000, GF(11) ) );
11
gap> Size( NordstromRobinsonCode() );
256
```

```
Dimension( <C> )
```

Dimension devuelve el parámetro k de **C**, es decir, la dimensión del código o el número de bits de información. Esta función no está definido para códigos no lineales.

```
gap> Dimension( NordstromRobinsonCode() );
Error, dimension is only defined for linear codes
gap> Dimension( NullCode( 5, GF(5) ) );
0
gap> C := BCHCode( 15, 4, GF(4) );
a cyclic [15,7,5]4..8 BCH code, delta=5, b=1 over GF(4)
gap> Dimension( C );
7
gap>LeftActingDomain( C );
GF(2^2)
```

```
AsSSortedList( C )
```

AsSSortedList devuelve una lista con los elementos de **C** ordenados.

```
 gap> C := ConferenceCode( 5 );
a (5,12,2)1..4 conference code over GF(2)
gap> AsSSortedList( C );
[ [ 0 0 0 0 0 ], [ 0 0 1 1 1 ], [ 0 1 0 1 1 ], [ 0 1 1 0 1 ], [ 0 1 1 1 0 ],
  [ 1 0 0 1 1 ], [ 1 0 1 0 1 ], [ 1 0 1 1 0 ], [ 1 1 0 0 1 ], [ 1 1 0 1 0 ],
  [ 1 1 1 0 0 ], [ 1 1 1 1 1 ] ]
gap> CodewordNr( C, [ 1, 2 ] );
[ [ 0 0 0 0 0 ], [ 0 0 1 1 1 ] ]
```

```
 CodewordNr( C, list )
```

CodewordNr devuelve una palabra o palabras del cdigo.

```
 gap> R := ReedSolomonCode(2,2);
a cyclic [2,1,2]1 Reed-Solomon code over GF(3)
gap> AsSSortedList(R);
[ [ 0 0 ], [ 1 1 ], [ 2 2 ] ]
gap> CodewordNr(R, [1,3]);
[ [ 0 0 ], [ 2 2 ] ]
gap> C := HadamardCode( 16 );
a (16,32,8)5..6 Hadamard code of order 16 over GF(2)
gap> AsSSortedList(C)[17] = CodewordNr( C, 17 );
true
```

1.14. Grabando códigos

```
Save( <filename>, <C>, <varname> )
```

Save graba el código **C** en el fichero **filename**. Si el fichero no existe, entonces se crea,y si ya existe entonces machaca la información que contiene. El código grabado recibe el nombre **varname** y puede ser recuperado con **Read(filename)**.

```
gap> C1 := HammingCode( 4, GF(3) );
a linear [40,36,3]1 Hamming (4,3) code over GF(3)
gap> Save( "mycodes.lib", C1, "Ham_4_3");
gap> Read( "mycodes.lib" ); Ham_4_3;
a linear [40,36,3]1 Hamming (4,3) code over GF(3)
gap> Ham_4_3 = C1;
true
```

1.15. Matriz generadora

```
GeneratorMat( <C> )
```

GeneratorMat devuelve la matriz generadora de **C**. El código consistirá en todas las combinaciones lineales de las filas de la matriz. Si **C** no es lineal, entonces la función devuelve un error.

```
 gap> GeneratorMat( HammingCode( 3, GF(2) ) );
[ <an immutable GF2 vector of length 7>, <an immutable GF2 vector of length 7>
    , <an immutable GF2 vector of length 7>,
  <an immutable GF2 vector of length 7> ]
 gap> Display(GeneratorMat( HammingCode( 3, GF(2) ) ));
 1 1 1 . . . .
 1 . . 1 1 . .
 . 1 . 1 . 1 .
 1 1 . 1 . . 1

 gap> Print(GeneratorMat( HammingCode( 3, GF(2) ) ));
[ [ Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
  [ Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2) ],
  [ 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2) ],
  [ Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0 ] ]

 gap> GeneratorMat( RepetitionCode( 5, GF(25) ) );
    [ [ Z(5)^0, Z(5)^0, Z(5)^0, Z(5)^0, Z(5)^0 ] ]
 gap> GeneratorMat( NullCode( 14, GF(4) ) );
    [  ]
 gap> GeneratorMat( ElementsCode( [[0, 0, 1 ], [1, 1, 0 ]], GF(2) ));
    Error, non-linear codes don't have a generator matrix
|
```

1.16. Matriz de paridad

```
CheckMat( <C> )
```

CheckMat devuelve la matriz de paridad del código. La traspuesta de esta matriz es la inversa a derecha de la matriz generadora. Esta matriz es calculada a partir de la matriz generadora, el polinomio generador o el polinomio de chequeo si ya ha sido calculado.

```
 gap> CheckMat( HammingCode(3, GF(2) ) );
[ <an immutable GF2 vector of length 7>, <an immutable GF2 vector of length 7>
```

```
    , <an immutable GF2 vector of length 7> ]
gap> CheckMat( RepetitionCode( 5, GF(25) ) );
[ [ Z(5)^0, Z(5)^2, 0*Z(5), 0*Z(5), 0*Z(5) ],
  [ 0*Z(5), Z(5)^0, Z(5)^2, 0*Z(5), 0*Z(5) ],
  [ 0*Z(5), 0*Z(5), Z(5)^0, Z(5)^2, 0*Z(5) ],
  [ 0*Z(5), 0*Z(5), 0*Z(5), Z(5)^0, Z(5)^2 ] ]
gap> CheckMat( WholeSpaceCode( 12, GF(4) ) );
[  ]
```

1.17. Polinomio generador y de chequeo

De la misma forma que en la sección anterior, se puede calcular el polinomio generador y de chequeo de un código dado. Si el código no es cíclico ambas funciones devuelven un error.

```
gap> GeneratorPol(GeneratorMatCode([[1, 1, 0], [0, 1, 1]], GF(2)));
Z(2)^0+x_1
gap> GeneratorPol( WholeSpaceCode( 4, GF(2) ) );
Z(2)^0
gap> GeneratorPol( NullCode( 7, GF(3) ) );
-Z(3)^0+x_1^7
```

```
gap> CheckPol(GeneratorMatCode([[1, 1, 0], [0, 1, 1]], GF(2)));
Z(2)^0+x_1+x_1^2
gap> CheckPol(WholeSpaceCode(4, GF(2)));
Z(2)^0+x_1^4
gap> CheckPol(NullCode(7,GF(3)));
Z(3)^0
```

1.18. Raices de un código

```
RootsOfCode( <C> )
```

RootsOfCode devuelve la lista de ceros del polinomio generador del código cíclico **C**.

En el proceso inverso, es posible construir un código a partir de su raices.

```
gap> C1 := ReedSolomonCode( 16, 5 );
a cyclic [16,12,5]3..4 Reed-Solomon code over GF(17)
```

```
gap> RootsOfCode( C1 );
[ Z(17), Z(17)^2, Z(17)^3, Z(17)^4 ]
gap> C2 := RootsCode( 16, last );
a cyclic [16,12,5]3..4 code defined by roots over GF(17)
gap> C1 = C2;
true
```

1.19. Otros parámetros

`WordLength( <C> )`

WordLength devuelve la longitud de palabra del código n. Los elementos de un código cíclico son polinomios de grado máximo $n-1$, y los calculos son realizados módulo $x^n - 1$.

```
gap> WordLength( NordstromRobinsonCode() );
16
gap> WordLength( PuncturedCode( WholeSpaceCode(7) ) );
6
gap> WordLength( UUVCode( WholeSpaceCode(7), RepetitionCode(7) ) );
14
```

`Redundancy( <C> )`

Redundancy devuelve la redundancia o bits de paridad r del código $\mathbf{C}$. Si $\mathbf{C}$ no es un código lineal devuelve un error

Un código lineal tiene dimensión k, longitud de palabra n, y redundancia $r = n - k$.

```
gap> C := TernaryGolayCode();
a cyclic [11,6,5]2 ternary Golay code over GF(3)
gap> Redundancy(C);
5
gap> Redundancy( DualCode(C) );
6
```

`MinimumDistance( <C> )`

MinimumDistance devuelve la mínima distancia de **C** . Para códigos lineales, la mínima distancia es igual al menor peso de todas las palabras del código. Si el código tiene un solo elemento, la distancia mínima es la longitud de la palabra código.

```
gap> C := MOLSCode(7);; MinimumDistance(C);
3
gap> WeightDistribution(C);
[ 1, 0, 0, 24, 24 ]
gap> MinimumDistance( WholeSpaceCode( 5, GF(3) ) );
1
gap> MinimumDistance( NullCode( 4, GF(2) ) );
4
gap> C := ConferenceCode(9);; MinimumDistance(C);
4
```

1.19.1. Distribución de pesos

```
WeightDistribution( <C> )
```

WeightDistribution devuelve la distribución de pesos de **C**. El i-ésimo elemento de vector contiene el número de elementos del código con peso $i - 1$.

```
gap> WeightDistribution( ConferenceCode(9) );
[ 1, 0, 0, 0, 0, 18, 0, 0, 0, 1 ]
gap> WeightDistribution( RepetitionCode( 7, GF(4) ) );
[ 1, 0, 0, 0, 0, 0, 0, 3 ]
gap> WeightDistribution( WholeSpaceCode( 5, GF(2) ) );
[ 1, 5, 10, 10, 5, 1 ]
```

1.19.2. CodeWeightEnumerator

```
CodeWeightEnumerator( <code> )
```

CodeWeightEnumerator devuelve la siguiente expresión :

$$f(x) = \sum_{i=0}^{n} A_i x^i, \tag{1.1}$$

donde A_i es el número de palabras del código con peso i.

```
gap> CodeWeightEnumerator( ElementsCode( [ [ 0,0,0 ], [ 0,0,1 ],
> [ 0,1,1 ], [ 1,1,1 ] ], GF(2) ) );
x^3 + x^2 + x + 1
gap> CodeWeightEnumerator( HammingCode( 3, GF(2) ) );
x^7 + 7*x^4 + 7*x^3 + 1
```

1.19.3. CodeDistanceEnumerator

```
CodeDistanceEnumerator( <code>, <word> )
```

CodeDistanceEnumerator devuelve un polinomio de la siguiente forma:

$$f(x) = \sum_{i=0}^{n} B_i x^i,$$

donde B_i es el número de palabras del código con distancia i respecto la palabras **word**.

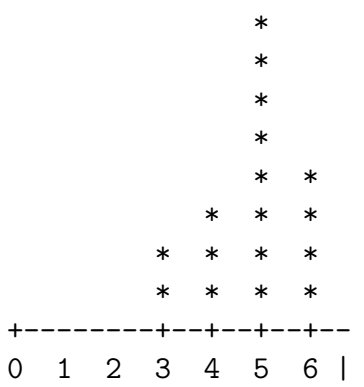
```
gap> CodeDistanceEnumerator( HammingCode( 3, GF(2) ),[0,0,0,0,0,0,1] );
x^6 + 3*x^5 + 4*x^4 + 4*x^3 + 3*x^2 + x
gap> CodeDistanceEnumerator( HammingCode( 3, GF(2) ),[1,1,1,1,1,1,1] );
x^7 + 7*x^4 + 7*x^3 + 1 # '[1,1,1,1,1,1,1]' $\in$ 'HammingCode( 3, GF(2 ) )'
```

1.20. Histograma de pesos

```
WeightHistogram( <C> )
WeightHistogram( <C>, <h> )
```

Esta función dibuja es histograma de la distribución de pesos del código **C**. **h** indica el tamaño de las columnas del histograma, cuyo valor por defecto es un tercio del tamaño de la pantalla.

```
gap> H := HammingCode(2, GF(5));
a linear [6,4,3]1 Hamming (2,5) code over GF(5)
gap> WeightDistribution(H);
[ 1, 0, 0, 80, 120, 264, 160 ]
gap> WeightHistogram(H);
264---------------
```



1.21. Codificación

La codificación en códigos lineales es muy sencilla. Sólo hay que hacerle corresponder a cada mensaje, representados con sus bits de información, una palabra código , que tendrá que ser una combinación de los elementos de la base o un multiplo del polinomio generador del código.

1.21.1. PermutedCols

```
PermutedCols( <M>, <P> )
```

PermutedCols devuelve la matriz **M** con las permutaciones indicadas en **P** realizadas sobre sus columnas.

```
gap> M := [[1,2,3,4],[1,2,3,4]];; PrintArray(M);
[ [  1,  2,  3,  4 ],
  [  1,  2,  3,  4 ] ]
gap> PrintArray(PermutedCols(M, (1,2,3)));
[ [  3,  1,  2,  4 ],
  [  3,  1,  2,  4 ] ] |
```

1.21.2. PutStandardForm

```
PutStandardForm( <M> )
PutStandardForm( <M>, <idleft> )
```

PutStandardForm pasa la matriz **M** a su forma estandar y devuelve las permutaciones necesarias para realizar dicha operación. **idleft** es un argumento

booleano. Si tiene valor *true*, la matriz identidad será colocada a la izquierda de la matriz, en caso contrario será colocada a la derecha.

La función **BaseMat** funciona de igual forma que la anterior función, pero nunca aplica operaciones sobre columnas, operación que si puede tener efecto con **PutStandarForm**.

```
gap> M := Z(2)*[[1,0,0,1],[0,0,1,1]];; PrintArray(M);
[ [  Z(2)^0,  0*Z(2),  0*Z(2),  Z(2)^0 ],
  [  0*Z(2),  0*Z(2),  Z(2)^0,  Z(2)^0 ] ]
gap> PutStandardForm(M);                  # identity at the left side
(2,3)
gap> PrintArray(M);
[ [  Z(2)^0,  0*Z(2),  0*Z(2),  Z(2)^0 ],
  [  0*Z(2),  Z(2)^0,  0*Z(2),  Z(2)^0 ] ]
gap> PutStandardForm(M, false);           # identity at the right side
(1,4,3)
gap> PrintArray(M);
[ [  0*Z(2),  Z(2)^0,  Z(2)^0,  0*Z(2) ],
  [  0*Z(2),  Z(2)^0,  0*Z(2),  Z(2)^0 ] ]
```

1.21.3. IsInStandardForm

```
IsInStandardForm( <M> )
IsInStandardForm( <M>, <idleft> )
```

IsInStandardForm devuelve *true* si la matriz **M** está en su forma estandar o *false* en el caso contrario. Los demás argumentos son los mismo que en la función **PutStandardForm**.

```
gap> IsInStandardForm(IdentityMat(7, GF(2)));
true
gap> IsInStandardForm([[1, 1, 0], [1, 0, 1]], false);
true
gap> IsInStandardForm([[1, 3, 2, 7]]);
true
gap> IsInStandardForm(HadamardMat(4));
false
```

1.21.4. Matriz Generadora

```
GeneratorMatCode( <G> [, <Name> ], <F> )
```

GeneratorMatCode devuelve un código lineal con matriz generadora **G**. **G** debe ser una matriz sobre el cuerpo finito **F**. **Name** es una breve descripción del código.

```
gap> G := Z(3)^0 * [[1,0,1,2,0],[0,1,2,1,1],[0,0,1,2,1]];;
gap> C1 := GeneratorMatCode( G, GF(3) );
a linear [5,3,1..2]1..2 code defined by generator matrix over GF(3)
gap> C2 := GeneratorMatCode( IdentityMat( 5, GF(2) ), GF(2) );
a linear [5,5,1]0 code defined by generator matrix over GF(2)
```

1.22. Decodificación

```
Decode( <C>, <c> )
```

Decode decodifica la palabra **c** respecto el código **C**. **c** es una palabra código o una lista de palabras código. Si el registro con el cual se representa al código, tiene un campo llamado *specialDecoder*, entonces a la hora de decodificar la palabra en cuestión, corrigiendo los errores presentes, usa el algoritmo especial especificado en dicho campo. En caso contrario, utiliza el algoritmo basado en sindromes. En el primer caso, está incluidos código como los códigos Hamming y BCH.

Un decodificador especial puede ser creado definiendo una función

```
C!.specialDecoder := function(C, c) ... end;
```

Esta función tiene como argumentos a **C**, es decir, al mismo registro, y un vector del mismo tipo que las palabras códigos de **C**.

```
gap> C := HammingCode(3);
a linear [7,4,3]1 Hamming (3,2) code over GF(2)
gap> c := "1010"*C;                    # encoding
[ 1 0 1 1 0 1 0 ]
gap> Decode(C, c);                     # decoding
[ 1 0 1 0 ]
gap> Decode(C, Codeword("0010101"));
[ 1 1 0 1 ]                            # one error corrected
gap> C!.SpecialDecoder := function(C, c)
> return NullWord(Dimension(C));
```

```
> end;
function ( C, c ) ... end
gap> Decode(C, c);
[ 0 0 0 0 ]           # new decoder always returns null word
```

1.22.1. Matriz de paridad

```
CheckMatCode( <H> [, <Name> ], <F> )
```

CheckMatCode devuelve un código lineal definido a partir de la matriz de paridad. Si $\mathbf{H}$ es una $r \times n$-matriz, el código tiene palabras de longitud n, redundancia r y dimensión $n - r$.

```
gap> G := Z(3)^0 * [[1,0,1,2,0],[0,1,2,1,1],[0,0,1,2,1]];;
gap> C1 := CheckMatCode( G, GF(3) );
a linear [5,2,1..2]2..3 code defined by check matrix over GF(3)
gap> CheckMat(C1);
[ [ Z(3)^0, 0*Z(3), Z(3)^0, Z(3), 0*Z(3) ],
  [ 0*Z(3), Z(3)^0, Z(3), Z(3)^0, Z(3)^0 ],
  [ 0*Z(3), 0*Z(3), Z(3)^0, Z(3), Z(3)^0 ] ]
gap> C2 := CheckMatCode( IdentityMat( 5, GF(2) ), GF(2) );
a linear [5,0,5]5 code defined by check matrix over GF(2) |
```

1.22.2. Código dual

```
DualCode( <C> )
```

DualCode devuelve el código dual de $\mathbf{C}$. Este código estará formado por todas las palabras que son ortogonales a las palabras código de $\mathbf{C}$. Si $\mathbf{C}$ es lineal, su dual también lo será y su matriz generadora será la matriz de paridad de $\mathbf{C}$. En el caso de que $\mathbf{C}$ sea cíclico con polinomio generador $g(x)$, su dual tendrá como polinomio generador el recíproco del polinomio de chequeo de $\mathbf{C}$. Si el dual de un código es el mismo entonces se denomina *Self Dual*.

```
gap> R := ReedMullerCode( 1, 3 );
a linear [8,4,4]2 Reed-Muller (1,3) code over GF(2)
gap> RD := DualCode( R );
a linear [8,4,4]2 Reed-Muller (1,3) code over GF(2)
```

```
gap> R = RD;
true
gap> N := WholeSpaceCode( 7, GF(4) );
a cyclic [7,7,1]0 whole space code over GF(4)
gap> DualCode( N ) = NullCode( 7, GF(4) );
true
```

1.22.3. Array estándar

```
StandardArray( <C> )
```

StandardArray devuelve el array estandar asociada al código **C**. El array estandar es una tabla formado por q^r filas y q^k columnas, donde q es el tamaño del alfabeto o cuerpo base, r la redundancia y k la dimensión del código. La primera fila son los elementos del código, y las restantes son palabras que no pertenecen al código y que se obtienen sumando un error por fila, a todas las palabras del código. Si el código no es lineal, la función devuelve un error.

```
gap> StandardArray(RepetitionCode(3,GF(3)));
[ [ [ 0 0 0 ], [ 1 1 1 ], [ 2 2 2 ] ],
  [ [ 0 0 1 ], [ 1 1 2 ], [ 2 2 0 ] ],
  [ [ 0 0 2 ], [ 1 1 0 ], [ 2 2 1 ] ],
  [ [ 0 1 0 ], [ 1 2 1 ], [ 2 0 2 ] ],
  [ [ 0 2 0 ], [ 1 0 1 ], [ 2 1 2 ] ],
  [ [ 1 0 0 ], [ 2 1 1 ], [ 0 2 2 ] ],
  [ [ 1 2 0 ], [ 2 0 1 ], [ 0 1 2 ] ],
  [ [ 2 0 0 ], [ 0 1 1 ], [ 1 2 2 ] ],
  [ [ 2 1 0 ], [ 0 2 1 ], [ 1 0 2 ] ] ]
```

1.22.4. Síndromes

```
Syndrome( <C>, <c> )
```

Syndrome devuelve el síndrome de la palabra **c** respecto el código **C**. Si **c** es un elemento del código, el síndrome es cero.

El síndrome no está definido para códigos no lineales. En ese caso devuelve un error.

```
gap> C := HammingCode(4);
```

```
a linear [15,11,3]1 Hamming (4,2) code over GF(2)
gap> v := CodewordNr( C, 7 );
[ 0 0 0 0 0 0 0 0 1 1 0 0 1 1 0 ]
gap> Syndrome( C, v );
[ 0 0 0 0 ]
gap> Syndrome( C, "000000001100111" );
[ 1 1 1 1 ]
gap> Syndrome( C, "000000000000001" );
[ 1 1 1 1 ]    # the same syndrome\: both codewords are in the same
               # coset of C
```

1.22.5. Tabla de síndromes

```
SyndromeTable( <C> )
```

SyndromeTable devuelve la tabla de síndromes asociada al código **C**. Esta tabla está formado por dos columnas, una primera formado por los errores de menor peso que se pueden corregir, y una segunda formado por los sindromes de esos errores.

```
gap> H := HammingCode(2);
a linear [3,1,3]1 Hamming (2,2) code over GF(2)
gap> SyndromeTable(H);
[ [ [ 0 0 0 ], [ 0 0 ] ], [ [ 1 0 0 ], [ 0 1 ] ],
  [ [ 0 1 0 ], [ 1 0 ] ], [ [ 0 0 1 ], [ 1 1 ] ] ]
gap> c := Codeword("101");
[ 1 0 1 ]
gap> c in H;
false          # c is not an element of H
gap> Syndrome(H,c);
[ 1 0 ]        # according to the syndrome table,
               # the error vector [ 0 1 0 ] belongs to this syndrome
gap> c - Codeword("010") in H;
true           # so the corrected codeword is
               # [ 1 0 1 ] - [ 0 1 0 ] = [ 1 1 1 ],
               # this is an element of H
```

1.23. Códigos Hamming

```
HammingCode( <r>, <F> )'
```

HammingCode devuelve un código Hamming con redundancia **r**. Un código Hamming es capaz de corregir errores de peso 1, y las columnas de la matriz de paridad son todas los posibles vectores de longitud **r**, eliminandos los vectores que son multiplos de otro.

Si q es el tamaño del cuerpo base **F**, el código resultante es un código lineal con parámetros $[(q^{<r>}-1)/(q-1),(q^{<r>}-1)/(q-1)-<r>,3]$.

```
gap> C1 := HammingCode( 4, GF(2) );
a linear [15,11,3]1 Hamming (4,2) code over GF(2)
gap> C2 := HammingCode( 3, GF(9) );
a linear [91,88,3]1 Hamming (3,9) code over GF(9) |
```

1.24. Código de ReedMuller

```
ReedMullerCode( <r>, <k> )
```

ReedMullerCode devuelve un código binario $R(<r>,<k>)$ con dimensión **k** y orden **r**. Este es un código con longitud $2^{<k>}$ y mínima distancia $2^{<k>-<r>}$. Por definición, un código de ReedMuller de orden r de longitud $n=2^m$, con $0 \leq r \leq m$, es el conjunto de todos los vectores f, con f una función booleana representada por un polinomio de grado r.

.

```
gap> ReedMullerCode( 1, 3 );
a linear [8,4,4]2 Reed-Muller (1,3) code over GF(2)
```

1.25. Códigos aleatorios

```
RandomLinearCode( <n>, <k> , <F> )
```

RandomLinearCode devuelve un código lineal aleatorio con longitud de palabra n, dimensión k y cuerpo base F.

```
gap> C := RandomLinearCode( 15, 4, GF(3) );
a linear [15,4,1..4]6..10 random linear code over GF(3)
```

```
gap> RandomSeed( 13 ); C1 := RandomLinearCode( 12, 5, GF(5) );
a linear [12,5,1..5]4..7 random linear code over GF(5)
gap> RandomSeed( 13 ); C2 := RandomLinearCode( 12, 5, GF(5) );
a linear [12,5,1..5]4..7 random linear code over GF(5)
gap> C1 = C2;
true    # Thanks to RandomSeed
```

1.26. Mejor código lineal conocido

```
BestKnownLinearCode( <n>, <k> , <F> )
```

BestKnownLinearCode devuelve el mejor código lineal conocido de longitud n, dimensión k y cuerpo base F.

```
gap> C1 := BestKnownLinearCode( 23, 12, GF(2) );
a cyclic [23,12,7]3 binary Golay code over GF(2)
gap> C1 = BinaryGolayCode();
true
gap> Display( BestKnownLinearCode( 8, 4, GF(4) ) );
a linear [8,4,4]2..3 U|'\|'|U+V construction code of
U: a cyclic [4,3,2]1 dual code of
   a cyclic [4,1,4]3 repetition code over GF(4)
V: a cyclic [4,1,4]3 repetition code over GF(4)
gap> C := BestKnownLinearCode(131,47);
a linear [131,47,28..32]23..68 shortened code |
```

1.27. Códigos cíclicos

Los elementos de un código cíclico C son múltiplos del polinomio generador $g(x)$, donde todos los cálculos son realizados módulo $x^n - 1$. Por lo tanto, todos los elementos del código son polinomios de grado menor que n. Un código cíclico es un ideal en el anillo de polinomios módulo $x^n - 1$. El polinomio $g(x)$ es el *polinomio generador* de C y es el polinomio mónico de menor grado que genera a C. Además es un divisor de $x^n - 1$.

El *polinomio de chequeo* es definido como $h(x) = x^n - 1/g(x)$. Este polinomio también es un divisor de $x^n - 1$ y verifica que $c(x)h(x) = 0 \pmod{(x^n - 1)}$ para toda palabra $c(x) \in C$.

1.27.1. Generación de un código cíclico

```
GeneratorPolCode( <g>, <n> [, <Name> ], <F> )
```

GeneratorPolCode define el código cíclico con polinomio generador **g**, longitud de palabra **n**, y cuerpo base **F**. **g** puede ser un polinomio sobre **F**, o una lista de coeficientes sobre **F**, en cuyo caso los enteros son convertidos a **F**. **Name** puede contener una breve descripción del código.

Si **g** no es un divisor de $x^n - 1$, entonces no puede generar un código lineal con longitud de palabra **n**. En este caso, el código definido es un código cíclico con polinomio generador $gcd(g, x^n - 1)$.

```
gap> x:= Indeterminate( GF(2) );; P:= x^2+1;
Z(2)^0+x^2
gap> G := GeneratorPolCode(P, 7, GF(2));
a cyclic [7,6,1..2]1 code defined by generator polynomial over GF(2)
gap> GeneratorPol( G );
Z(2)^0+x
gap> G2 := GeneratorPolCode( x+1, 7, GF(2));
a cyclic [7,6,1..2]1 code defined by generator polynomial over GF(2)
gap> GeneratorPol( G2 );
Z(2)^0+x
```

1.27.2. Polinomio de chequeo

```
CheckPolCode( <h>, <n> [, <Name> ], <F> )
```

CheckPolCode define un código con polinomio de chequeo **h**, longitud de palabra **n** y cuerpo base **F**. Respecto a los demás argumentos de esta función, se definen de la misma forma que el apartado anterior.

```
gap>  x:= Indeterminate( GF(3) );; P:= x^2+2;
-Z(3)^0+x_1^2
gap> H := CheckPolCode(P, 7, GF(3));
a cyclic [7,1,7]4 code defined by check polynomial over GF(3)
gap> CheckPol(H);
-Z(3)^0+x_1
gap> Gcd(P, X(GF(3))^7-1);
-Z(3)^0+x_1
```

1.27.3. Polinomio recíproco

```
ReciprocalPolynomial( <P> )
```

ReciprocalPolynomial devuleve el polinomio recíproco de **P**. Este polinomio posee los mismo coeficientes que **P** pero en orden inverso. Por lo tanto si $P = a_0 + a_1X + ... + a_nX^n$, su polinomio recíproco será $P' = a_n + a_{n-1}X + ... + a_0X^n$.

```
ReciprocalPolynomial( <P> , <n> )
```

En este caso especificamos que el grado mínimo de **P** es **n**, y si el grado de **P** es menor se completa con ceros.

```
gap> P := UnivariatePolynomial( GF(3), Z(3)^0 * [1,0,1,2] );
Z(3)^0+x_1^2-x_1^3
gap> ReciprocalPolynomial( P, 6 );
-x_1^3+x_1^4+x_1^6
```

1.27.4. RootsCode

```
RootsCode( <n>, <list> )
```

Esta función define un código con longitud de palabra **n** y donde todas sus palabras tienen como ceros a todos elementos del cuerpo especificados en **list**. Todos los elementos de esta lista son enésimas raices de la unidad en la extensión de cuerpo $GF(q^m)$. El código resultante está definido sobre $GF(q)$ y es el mayor código cíclico donde **list** es un subconjunto de todas sus raices.

```
gap> a := PrimitiveUnityRoot( 3, 14 );
Z(3^6)^52
gap> C1 := RootsCode( 14, [ a^0, a, a^3 ] );
a cyclic [14,7,3..6]3..7 code defined by roots over GF(3)
gap> MinimumDistance( C1 );
4
gap> b := PrimitiveUnityRoot( 2, 15 );
Z(2^4)
gap> C2 := RootsCode( 15, [ b, b^2, b^3, b^4 ] );
a cyclic [15,7,5]3..5 code defined by roots over GF(2)
gap> C2 = BCHCode( 15, 5, GF(2) );
true
```

```
RootsCode( <n>, <list>, <F> )
```

De esta forma, **list** está formado por un conjunto de enteros entre 0 y $n-1$. El código resultante es definido sobre **F**. **GUAVA** calcula la enésima raiz de la unidad, α, en la extensión del cuerpo **F** y en **list** especificaremos el conjunto de potencias de α como los ceros del código.

```
gap> C := RootsCode( 4, [ 1, 2 ], GF(5) );
a cyclic [4,2,3]2 code defined by roots over GF(5)
gap> RootsOfCode( C );
[ Z(5), Z(5)^2 ]
gap> C = ReedSolomonCode( 4, 3 );
true
```

1.27.5. Raiz primitiva de la unidad

```
PrimitiveUnityRoot( <F>, <n> )
```

PrimitiveUnityRoot devuelve la enésima raiz de la unidad en una extensión del cuerpo **F**. Este elemento verifica que $a^n = \text{mod } n$, y n es el menor entero que verifica esa condición.

```
gap> PrimitiveUnityRoot( GF(2), 15 );
Z(2^4)
gap> last^15;
Z(2)^0
gap> PrimitiveUnityRoot( GF(8), 21 );
Z(2^6)^3
```

1.27.6. Lista de códigos cíclicos

```
CyclicCodes( <n>, <F> )
```

CyclicCodes devuelve una lista con todos los códigos cíclicos de longitud **n** sobre **F**.

```
NrCyclicCodes( <n>, <F> )
```

Esta función devuelve el número de códigos cíclicos de longitud **n** sobre **F**.

```
gap> NrCyclicCodes( 23, GF(2) );
8
gap> codelist := CyclicCodes( 23, GF(2) );
[ a cyclic [23,23,1]0 enumerated code over GF(2),
  a cyclic [23,22,1..2]1 enumerated code over GF(2),
  a cyclic [23,11,1..8]4..7 enumerated code over GF(2),
  a cyclic [23,0,23]23 enumerated code over GF(2),
  a cyclic [23,11,1..8]4..7 enumerated code over GF(2),
  a cyclic [23,12,1..7]3 enumerated code over GF(2),
  a cyclic [23,1,23]11 enumerated code over GF(2),
  a cyclic [23,12,1..7]3 enumerated code over GF(2) ]
gap> BinaryGolayCode() in codelist;
true
gap> RepetitionCode( 23, GF(2) ) in codelist;
true
```

1.28. Códigos BCH

```
BCHCode( <n>, <d> , <F> )
BCHCode( <n>, <b>, <d>, <F> )
```

Esta función define un *Bose-Chaudhuri-Hockenghem* código. Este código es el mayor código cíclico posible de longitud **n** sobre el cuerpo **F**, con polinomio generador con ceros

$$a^{<b>}, a^{<b>+1}, ..., a^{<b>+<d>-2},$$

donde a es la enésima raiz de la unidad en $GF(q^m)$, **b** es un entero mayor que 1 y m es el orden multiplicativo de q módulo n. **b** vale por defecto 1, y n y q deben ser coprimos. El polinomio generador de este código es igual al producto de los polinomios mínimos de los ceros anteriores $X^{<b>}, X^{<b>+1}, ..., X^{<b>+<d>-2}$.

```
gap> C1 := BCHCode( 15, 3, 5, GF(2) );
a cyclic [15,5,7]5 BCH code, delta=7, b=1 over GF(2)
gap> C1.designedDistance;
7
gap> C2 := BCHCode( 23, 2, GF(2) );
a cyclic [23,12,5..7]3 BCH code, delta=5, b=1 over GF(2)
gap> C2.designedDistance;
5
gap> MinimumDistance(C2);
7
```

1.29. Códigos de Reed-Solomon

```
ReedSolomonCode( <n>, <d> )
```

ReedSolomonCode devueve un código de longitud de palabra **n** y distancia mínima. No es más que un código *BCH* sobre el cuerpo $GF(q)$, con $q = n + 1$, dimensión $n - d + 1$.

```
gap> C1 := ReedSolomonCode( 3, 2 );
a cyclic [3,2,2]1 Reed-Solomon code over GF(4)
gap> C2 := ReedSolomonCode( 4, 3 );
a cyclic [4,2,3]2 Reed-Solomon code over GF(5)
gap> RootsOfCode( C2 );
[ Z(5), Z(5)^2 ]
gap> IsMDSCode(C2);
true
```

1.30. Manipulación de códigos

1.30.1. ExtendedCode

```
'ExtendedCode( <C> [, <i> ] )'
```

ExtendedCode extiende el código **C i** veces. **i** vale 1 por defecto y la extensión del código consiste en sumar un bit de paridad después de la última coordenada. La longitud del código aumenta tantas veces como indique **i** y las palabras del nuevo código verifican que la suma de todas sus coordenadas es cero (módulo el cuerpo base de C). Si estamos en el caso binario, todas las palabras del código extendido tendrán peso par y la mínima distancia será $d + 1$ si el código es par, es decir, si todas sus palabras tienen peso par.

```
gap> C1 := HammingCode( 3, GF(2) );
a linear [7,4,3]1 Hamming (3,2) code over GF(2)
gap> C2 := ExtendedCode( C1 );
a linear [8,4,4]2 extended code
gap> IsEquivalent( C2, ReedMullerCode( 1, 3 ) );
true
gap> List( Elements( C2 ), WeightCodeword );
[ 0, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 8 ]
gap> PuncturedCode( C2 ) = C1;
```

```
true
gap> C3 := EvenWeightSubcode( C1 );
a linear [7,3,4]2..3 even weight subcode
```

1.30.2. PuncturedCode

```
PuncturedCode( <C> )
```

PuncturedCode genera un nuevo código eliminando la última coordenada o posición de todas las palabras de **C**. La nueva longitud de palabra disminuye en una unidad y la mínima distancia normalmente también se decrementa en una unidad.

```
PuncturedCode( <C>, <L> )
```

De esta forma se obtiene un nuevo código eliminando todas las posiciones especificadas en **L**.

```
gap> C1 := BCHCode( 15, 5, GF(2) );
a cyclic [15,7,5]3..5 BCH code, delta=5, b=1 over GF(2)
gap> C2 := PuncturedCode( C1 );
a linear [14,7,4]3..5 punctured code
gap> ExtendedCode( C2 ) = C1;
false
gap> PuncturedCode( C1, [1,2,3,4,5,6,7] );
a linear [8,7,1..2]1 punctured code
gap> PuncturedCode( WholeSpaceCode( 4, GF(5) ) );
a linear [3,3,1]0 punctured code  # The dimension decreased from 4 to 3 |
```

1.30.3. PermutedCode

```
PermutedCode( <C>, <L> )
```

PermutedCode devuelve un nuevo código como resultado de permutar las posiciones de todas las palabras de código **C** tal como se indica en **L**.

```
gap> C1 := PuncturedCode( ReedMullerCode( 1, 4 ) );
a linear [15,5,7]5 punctured code
```

```
gap> C2 := BCHCode( 15, 7, GF(2) );
a cyclic [15,5,7]5 BCH code, delta=7, b=1 over GF(2)
gap> C2 = C1;
false
gap> p := CodeIsomorphism( C1, C2 );
( 2,13, 7,10, 8, 3, 5, 4,14)(12,15)
gap> C3 := PermutedCode( C1, p );
a linear [15,5,7]5 permuted code
gap> C2 = C3;
true
```

1.30.4. ExpurgatedCode

```
ExpurgatedCode( <C>, <L> )
```

ExpurgatedCode elimina las palabras de **C** que son especificadas en **L**. **C** debe ser lineal y el nuevo código también lo será.

```
gap> C1 := HammingCode( 4 );; WeightDistribution( C1 );
[ 1, 0, 0, 35, 105, 168, 280, 435, 435, 280, 168, 105, 35, 0, 0, 1 ]
gap> L := Filtered( Elements(C1), i -> WeightCodeword(i) = 3 );;
gap> C2 := ExpurgatedCode( C1, L );
a linear [15,4,3..4]5..11 code, expurgated with 7 word(s)
gap> WeightDistribution( C2 );
[ 1, 0, 0, 0, 14, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0 ]
```

Esta función no es válida para códigos no lineales. Si queremos realizar una operación similar para códigos no lineales debemos usar **RemovedElementsCode**.

1.30.5. AugmentedCode

```
AugmentedCode( <C>, <L> )
```

AugmentedCode realiza la operación contraria que la función anterior. Recuerde que **C** debe ser un código lineal y al igual que el resultado que devuelve.

```
gap> C31 := ReedMullerCode( 1, 3 );
a linear [8,4,4]2 Reed-Muller (1,3) code over GF(2)
gap> C32 := AugmentedCode(C31,["00000011","00000101","00010001"]);
```

```
a linear [8,7,1..2]1 code, augmented with 3 word(s)
gap> C32 = ReedMullerCode( 2, 3 );
true
```

Debemos utilizar la función **AddedElementsCode** para el caso no lineal.

1.30.6. ShortenedCode

```
ShortenedCode( <C> )
```

ShortenedCode devuelve el código **C** después de eliminar todas las palabras que empiezan por algo distinto de cero y quitando la primera posición o el primer bit de todas ellas. Si **C** es un código lineal con parámetros $[n, k, d]$, el nuevo código será otro código lineal con parámetros $[n-1, k-1, d]$. Es posible que la dimensión sea la misma y que la distancia mínima se incremente.

```
gap> C1 := HammingCode( 4 );
a linear [15,11,3]1 Hamming (4,2) code over GF(2)
gap> C2 := ShortenedCode( C1 );
a linear [14,10,3]2 shortened code
```

Si el código no es lineal, esta función chequea que dígito se repite con más frecuencia en la primera columna y elimina todas las palabras que no empiezan por él, después elimina la primera columna. El código resultante tendrá como mínimo la misma distancia mínima.

```
gap> C1 := ElementsCode( ["1000", "1101", "0011" ], GF(2) );
a (4,3,1..4)2 user defined unrestricted code over GF(2)
gap> MinimumDistance( C1 );
2
gap> C2 := ShortenedCode( C1 );
a (3,2,2..3)1..2 shortened code
gap> Elements( C2 );
[ [ 0 0 0 ], [ 1 0 1 ] ]
```

```
ShortenedCode( <C>, <L> )
```

En este caso realiza la misma operación pero mirando en todas las columnas especificadas en **L**.

```
gap> C1 := HammingCode( 5, GF(2) );
a linear [31,26,3]1 Hamming (5,2) code over GF(2)
gap> C2 := ShortenedCode( C1, [ 1, 2, 3 ] );
a linear [28,23,3]2 shortened code
gap> OptimalityLinearCode( C2 );
0
```

1.30.7. ConversionFieldCode

```
ConversionFieldCode( <C> )
```

ConversionFieldCode devuelve un nuevo código resultante de cambiar el cuerpo base. Si el cuerpo base de **C** es 'GF(q^m)', el nuevo código tendrá como cuerpo base a 'GF(q)'. Cada símbolo en cada palabra será sustituido por una cadena de m símbolos dado por su representación en el nuevo cuerpo base. Si **C** es un código con parámetros (n, M, d_1), el código resultante tendrá como parámetros a (nm, M, d_2), donde $d_2 > d_1$.

```
gap> R := RepetitionCode( 4, GF(4) );
a cyclic [4,1,4]3 repetition code over GF(4)
gap> R2 := ConversionFieldCode( R );
a linear [8,2,4]3..4 code, converted to basefield GF(2)
gap> Size( R ) = Size( R2 );
true
gap> GeneratorMat( R );
[ [ Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0 ] ]
gap> GeneratorMat( R2 );
[ [ Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2) ],
  [ 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0 ] ]
```

1.30.8. CosetCode

```
CosetCode( <C>, <w> )
```

CosetCode devuelve la clase o coclase de **C** respecto a la palabra **w**. **w** debe ser una palabra código.

```
gap> H := HammingCode(3, GF(2));
a linear [7,4,3]1 Hamming (3,2) code over GF(2)
gap> c := Codeword("1011011");; c in H;
false
gap> C := CosetCode(H, c);
a (7,16,3)1 coset code
gap> List(Elements(C), el-> Syndrome(H, el));
[ [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ],
  [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ],
  [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ] ]
# All elements of the coset have the same syndrome in H |
```

1.30.9. StandardFormCode

```
StandardFormCode( <C> )
```

StandardFormCode devuelve el mismo código en su forma estandar. Si **C** es lineal, la matriz generadora y de paridad reescritas en su forma estandar. Si las operaciones afectan a las columnas el nuevo se obtendrá el mismo código, en caso contrario el código será equivalente al de partida. **GUAVA** intenta hacerlo siempre que sea posible de la primera forma. De no poder ser asi, las permutaciones de las columnas pueden ser estudiadas usando **Display**.

Si **C** es cíclico, la matriz generadora no puede ser cambiada y esta función devolverá una copia de **C**.

```
gap> G := GeneratorMatCode( Z(2) * [ [0,1,1,0], [0,1,0,1], [0,0,1,1] ],
> "random form code", GF(2) );
a linear [4,2,1..2]1..2 random form code over GF(2)
gap> Codeword( GeneratorMat( G ) );
[ [ 0 1 0 1 ], [ 0 0 1 1 ] ]
gap> Codeword( GeneratorMat( StandardFormCode( G ) ) );
[ [ 1 0 0 1 ], [ 0 1 0 1 ] ] |
```

1.30.10. Suma directa de códigos

```
DirectSumCode( <C1>, <C2> )
```

DirectSumCode devuelve la suma directa de los códigos especificados en sus argumentos. Las palabras del nuevo código resultan de la concatenación de todas las palabras de **C1** con las de **C2**. Si C_i es un código con parámetros (n_i, M_i, d_i), el nuevo código tendrá como parámetros a $(n_1 + n_2, M_1 M_2, min(d_1, d_2))$. Si ambos códigos son lineales, el nuevo código también lo será, y si uno de ellos no lo es, el resultado tampoco lo será.

```
gap> C1 := ElementsCode( [ [1,0], [4,5] ], GF(7) );;
gap> C2 := ElementsCode( [ [0,0,0], [3,3,3] ], GF(7) );;
gap> D := DirectSumCode(C1, C2);;
gap> Elements(D);
[ [ 1 0 0 0 0 ], [ 1 0 3 3 3 ], [ 4 5 0 0 0 ], [ 4 5 3 3 3 ] ]
gap> D = C1 + C2;   # addition = direct sum
true
```

1.30.11. Producto directo de códigos

```
DirectProductCode( <C1>, <C2> )
```

DirectProductCode devuelve el producto directo de dos códigos lineales, **C1** y **C2** . Si C_i tiene matriz generadora G_i, el producto directo de C_1 y C_2 será un código lineal con matriz generadora el producto de *Kronecker* de G_1 y G_2 .

Si C_i es un código lineal con parámetros $[n_i, k_i, d_i]$, entonces su producto directo será un código lineal con parámetros $[n_1 n_2, k_1 k_2, d_1 d_2]$.

```
gap> L1 := LexiCode(10, 4, GF(2));
a linear [10,5,4]2..4 lexicode over GF(2)
gap> L2 := LexiCode(8, 3, GF(2));
a linear [8,4,3]2..3 lexicode over GF(2)
gap> D := DirectProductCode(L1, L2);
a linear [80,20,12]20..45 direct product code
```

1.30.12. Intersección de códigos

```
IntersectionCode( <$C_1$>, <$C_2$> )
```

IntersectionCode devuelve la intersección de los códigos que son especificados como argumentos. Si ambos códigos son lineales el resultado también será un código lineal. Y sin son cíclicos el resultado también será cíclico.

```
gap> C := CyclicCodes(7, GF(2));
[ a cyclic [7,7,1]0 enumerated code over GF(2),
  a cyclic [7,6,1..2]1 enumerated code over GF(2),
  a cyclic [7,3,1..4]2..3 enumerated code over GF(2),
  a cyclic [7,0,7]7 enumerated code over GF(2),
  a cyclic [7,3,1..4]2..3 enumerated code over GF(2),
  a cyclic [7,4,1..3]1 enumerated code over GF(2),
  a cyclic [7,1,7]3 enumerated code over GF(2),
  a cyclic [7,4,1..3]1 enumerated code over GF(2) ]
gap> IntersectionCode(C[6], C[8]) = C[7];
true
```

1.30.13. Union de códigos

```
UnionCode( <C1>, <C2> )
```

UnionCode devuelve la unión de dos códigos lineales y será un código lineal definido como la unión de todas sus palabras código y todas sus combinaciones lineales.

```
gap> G := GeneratorMatCode([[1,0,1],[0,1,1]]*Z(2)^0, GF(2));
a linear [3,2,1..2]1 code defined by generator matrix over GF(2)
gap> H := GeneratorMatCode([[1,1,1]]*Z(2)^0, GF(2));
a linear [3,1,3]1 code defined by generator matrix over GF(2)
gap> U := UnionCode(G, H);
a linear [3,3,1]0 union code
gap> c := Codeword("010");; c in G;
false
gap> c in H;
false
gap> c in U;
true
```