CONTRIBUTIONS to SCIENCE, **1** (2): 199-224 (1999)
Institut d'Estudis Catalans, Barcelona

OMEGA: a system for the effective construction, coding and decoding of block error-correcting codes

S. Xambó*
Departament de Matemàtica Aplicada i Facultat de Matemàtiques i Estadística, Universitat Politècnica de Catalunya

Abstract

In this work, we show how to implement effective constructions, coding and decoding of algebraic codes by means of Omega, a system specifically designed and programmed for general mathematical computations. For alternant codes, the main class we consider (which includes BCH, RS and classical Goppa codes), we give an implementation of the Euclidean division BM decoding algorithm. For cyclic codes we implement the Meggitt decoder, and to illustrate how it works we provide an implementation of the Meggitt syndrome tables for the two Golay codes. Finally, we present several other groups of functions and the computations and problems (still almost in the area of error-correcting codes) they solve.

Resum

L'objecte d'aquest treball és explicar com es poden implementar d'una manera efectiva, mitjançant el programa de manipulació simbòlica OMEGA, algunes de les construccions i operacions més importants de la teoria de codis correctors algebraics. Per als codis alternants, la classe més important que considerem, i que inclou els codis BCH, RS i de Goppa clàssics, presentem una implementació de l'algorisme de descodificació de Berlekamp-Massey. Per als codis cíclics, implementem l'algorisme de descodificació de Meggitt, i il·lustrem el seu funcionament, mitjançant la construcció de les corresponents taules de síndromes de Meggitt, per als codis de Golay. Finalment, presentem diversos altres grups de funcions, així com els càlculs i problemes (encara circumscrits gairebé a l'àrea de codis correctors) que ens permeten resoldre.

Effectively constructing, coding and decoding arbitrary block error-correcting codes –that is, by means of efficient computer programmes– turns out to be quite arduous. Indeed, Goppa codes, for example, involve at least the following objects:

- Arbitrary finite fields $\mathrm{F} = \mathrm{F}_q$.
- Polynomials and matrices over F.
- Algebraic curves X F.
- The points on X that are rational over F.
- Rational functions on X, their natural operations and their values at points of X.
- Divisors on X.

Furthermore, a system in which all these items, and the corresponding operations, can be represented, should also be able to express and run the diverse algorithms that typically appear in the area.

In this paper, we present the system OMEGA, with special attention to the services it provides to represent the objects and solve the problems mentioned above. In broad outline, after a first section in which we deal with some generalities about OMEGA, we devote the remaining sections to explain the key ideas of the theory of error-correcting codes that we need here, and, in each case, we will work out what is the bearing of OMEGA on them. Many other related services are introduced in the quite long chapter 1. In this way, many features of the programme OMEGA, and of the language interpreted by it (here called Ω) will gradually emerge. To be more specific, i section 2 we find some basic notions of block codes; in section 3, alternant codes, a class comprising BCH (with its important subclass RS) and the classical Goppa codes; in section 4, the Berlekamp-Massey (BM) decoding algorithm for alternant codes and its detailed implementation in OMEGA; in section 5, examples of how this decoder works; in section 6, which can be regarded as a more advanced continuation of section 1, we study some of the external functions that have been used in the previous sec-

* Author for correspondence: Sebastià Xambó, Departament de Matemàtica Aplicada II (MA2) i Facultat de Matemàtiques i Estadística (FME), Universitat Politècnica de Catalunya (UPC). Pau Gargallo, 5. Campus Sud-Edif. U. 08028 Barcelona, Catalonia (Spain). Tel. 34 93 401 69 26. Fax: 34 93 401 72 84. Email: sxd@ma2.upc.es.

Currently the author is president of the Catalan Society for Mathematics.

The OMEGA team 1998/1999, led by the author, consists of Daniel Marquès, Ramon Eixarch, Marc Castells, David Arso and Pere Garriga.

tions as well as other functions related to the computational aspects of error-correcting codes; in section 7, finally, we write down some conclusions and make a few additional remarks on the expressive power of Ω, including its multilingual features.

For more specialized applications in the coding area, see [26]; for other applications of OMEGA, see [27], the users manual, and [25], an OMEGA package for computations in intersection theory (two functions of this package are shown in section 6).

1. A first glimpse on OMEGA

In this section we provide a quick tour to some of the main features of the programme OMEGA (and of the language Ω). The reader interested in effective methods in coding can skim over this section, and may return to it later if needed.

OMEGA is written in C++. Here we run a version compiled with Visual C++ for Windows 98 called OMEGA/Athens/1999. This system should be useful for teaching several subjects (e.g. linear algebra, geometry, algebra, calculus, number theory, combinatorics, algebraic geometry) in the universities and for research on many fields of mathematics and physics.

The programme OMEGA/Athens has a command line of the form

```
User[n]: |
```

in which we can write Ω-expressions (n is a positive integer and | stands for a blinking cursor). For example,

```
User[127]: p = {i with i in 1..10000 suchthat
                prime?(i)}
```

assigns the list of prime numbers that are less than 10000 to the variable p. This expression is equivalent to

```
User[127]: p = {i suchthat prime?(i) with i in
                1..10000}
```

Anyhow, the answer is

```
OMEGA[127]:= {2,3,5,...,9973} :: List
User[128]: |
```

(the value of p is a list of 1229 integers and, in my laptop, OMEGA takes 0.17 s to answer). Now, the system is waiting for another expression. The dialog User-OMEGA will continue until we issue the command to exit, clicking the Close button in the File menu, or entering one of the following commands:

```
bye();
exit();
quit();
```

Types

The language Ω is *typed*. To indicate that a value x has type T, we (and OMEGA) write x :: T. Thus, the type of p above is `List` (by definition, a *list* is a finite sequence of objects enclosed within braces). The length of a list L can be obtained writing `dim(L)` or `length(L)`. The expression required to extract the i-th element of a list L is `L.i` (which results in an error message if i is not in the range of L).

The most basic type is `Integer` (written Z in the output). For example, `p.100` denotes the 100-th term in the list p, and we have:

```
User[128]: p.100;
OMEGA[128]:= 541 :: Z
```

Integers in Ω are roughly bounded to 4×10^{10} decimal digits. Thus, two digits of 2×10^{10} decimal digits can be multiplied. Or, theoretically, we could find n! for $n = 5\times10^{8}$!, although this would take too much time, even for a much lower n. For example, about 2 s are needed to calculate 20000! (66024 decimal digits), but it takes about 50 s to transform that number from the internal representation to the decimal form and print it on the screen.

Vectors and matrices

Vectors are similar to lists: a *vector* is a sequence of objects enclosed within brackets, e.g. [2,3,5,7], and the length of a vector v is given by `dim(v)` or `length(v)`. The difference between vectors and lists lies in the operations predefined for each type. For example, vectors of the same dimension can be added, and a vector can be multiplied by a scalar, but these operations are undefined for lists.

One remarkable operation with the vectors u and v (which also works for lists) is the concatenation `u|v`. Let us see two examples:

```
u = [1,2,3]; v = [a,b,c]

w=u|v
---> [1,2,3,a,b,c] :: Vector(Z[a,b,c])

u|-u|w|-v
--->
[1,2,3,-1,-2,-3,1,2,3,a,b,c,-a,-b,-c] ::
  Vector(Z[a,b,c])
```

A matrix is a sequence of vectors of the same length enclosed within brackets. The product of matrices, or of a matrix and a vector, is defined as usual. In addition, OMEGA has very powerful functions to construct and manipulate vectors and matrices. For example, if A and B are matrices with the same number of columns, `A & B` is the matrix obtained by stacking A on top of B. Likewise, if A and B have the same number of rows, `A | B` is the matrix whose rows are the concatenation of the corresponding rows of A and B. Let us also mention that the transpose of a matrix A is A′ (or `transpose(A)`). Some of these services will be used in the following sections. For further details, see [27].

Ranges

If a, b, c are integers, the type of the object a..b..c is Range (the object a..b coincides with a..b..1; note that in the construction of the list p at the beginning of this section we have used the range `1..10000`). The range r = a..b..c is a *latent*

way of referring to the integers a,a+c,a+2c,... that are not greater than b. We say 'latent' because these integers are not computed when the system evaluates r, but only when they are needed. If we write, for instance, [r], then we get the vector [a,a+c,a+2c,...]. Here is an example:

```
r=34..180..17;
---> 34..180..17 :: Range
[r];
---> [34, 51, 68, 85, 102, 119, 136, 153, 170]

                :: Vector(Z)
```

(hereafter, we will omit the prompt label `User[...]` and the results label `OMEGA[...]`, and we will only write the input expression and the resulting value with the symbol `--->` in between).

Functions

Functions in OMEGA have type Function. OMEGA has many built-in functions, or *internal* functions, like

```
binomial(n,m)
```

that yields the binomial number $\binom{n}{m}$,

```
prim(K)
```

that returns a primitive element of the (finite) field K (finite fields are treated in section), or

```
polirred(K,r,T)
```

that supplies us, given a finite field K, a positive integer r and an indeterminate T, with a monic irreducible polynomial of degree r with coefficients in K in the indeterminate T (an *indeterminate* is a name that has not been assigned a value).

The function `binomial(n,m)` has two arguments, which are supposed to be non-negative integers. The function `prim(K)` has a unique argument, which is a (finite) field. And the function `polirred(K,r,T)` has three arguments, which are, respectively, a finite field, a positive integer and an indeterminate.

Functions can also be defined by the user, or read from suitable files, and in this case we say that they are *external* functions. For example,

```
rv(n,m) := [random(m) with i in 1..n]
```

is an external function that constructs a vector of length n whose components are integers in the range 0..(m-1), each taken at random. Notice that here we use the internal function

```
random(m)
```

which returns, given a positive integer m, a pseudo-random integer in the range 0..(m-1). (We will use the function rv in the examples at the end of section 5.)

External functions like `rv(n,m)` can be defined at the prompt line, but it is usually more convenient to define several of them in a file which can be read (or loaded) with the function `read` (respectively `load`). We can call such a file an Ω-library. By default, its extension is assumed to be `.om` (`.omd` also works if no file `.om` can be found, but now this extension is considered obsolete). Thus,

```
read «tools»;
```

will read the expressions in the file `tools.om` (or `tools.omd` if `tools.om` does not exist), and in particular the function definitions therein. The expressions in a file are not displayed if the function used is `load`, but they are displayed sequentially if the function used is `read`. Files can be read using the `Open` option in the pull-down menu `File`.

Here is an example of a very simple Ω-library, consisting of only four functions (the symbol `#` opens a line comment):

```
# DEG-RAD.OMD
# To transform radians into degrees
rad2deg(r) := r*180/Pi;

# To transform degrees into radians
deg2rad(d) := d*Pi/180;

# Radians r into [d,m,s] --d degrees,
# m minutes, s seconds
rad2dms(r):=
  begin local x,d,m,s;
    x=rad2deg(r);   d=floor(x);
    # floor(x)=the integral part of x
    x=decimal(x)*60; m=floor(x);
    # decimal(x)=x-floor(x)
    x=decimal(x)*60; s=floor(x);
    [d,m,s]
  end;

# To find, if a=[d,m,s] is an angle in
# degrees, minutes, seconds,
# the corresponding number of radians.
dms2rad(a):=(a.1+a.2/60+a.3/3600)*Pi/180;
```

This file can be read with the command

```
read «deg-rad»
```

or with the command

```
load «deg-rad»
```

and we will then have the functions `rad2deg`, `deg2rad`, `rad2dms`, `dms2rad` at our disposal, as if they were internal functions. Note that `deg2rad` is an internal function that can be invoked with the usual ° symbol:

```
sin(30 °)
-> 0.500
```

Functions, as any other object in OMEGA, need not be bound to a name. Thus `x->1/x` stands for the function that maps a value x into its inverse $1/x$. If we want to give a name to this function, say f, we can do it in two ways. We can set

```
f(x):= 1/x
```

or just

```
f := x–>1/x
```

The value of both assignments is

```
x –> 1/x :: Function
```

and in both cases `f(3)` is `1/3`.

Functions can be arguments of other functions. For example, if v is a vector or a list, the function

```
inv(v) := map(x-->1/x,v)
```

yields the vector or list whose components are the inverses of the components of v:

```
inv([1..5])
---> [1,1/2,1/3,1/4,1/5] :: Vector(Q)
```

Functions f(x) of a single parameter can be called with the syntax

```
f x
```

(parenthesis are not needed). In the last example,

```
inv [1..5]
```

would mean the same. However, we usually write the parenthesis for clearness.

Polynomials

Univariate and polivaritate polynomials, with arbitrary coefficients, are supported by OMEGA. Let us see some examples. If a = [a_1,…,a_n] is a vector and T is an indeterminate, then

```
vector2pol(a,T)
```

returns the polynomial $a_1+a_2T+\ldots+a_nT^{n-1}$, whereas

```
roots2pol(a,T)
```

yields the polynomial $\Pi_{i=1}^{n}(T-a_i)$. This expression could also be written in OMEGA as follows:

```
product (T-a.i) with i in range(a)
```

If f is a polynomial and x an indeterminate, the expression

```
diff(f,x)
```

gives the derivative of f with respect to x.

```
diff(5*x^14+13*x^7*y^31,y);
---> 403 x^7 y^30
```

Finite fields

The OMEGA system has powerful functions for the creation and manipulation of finite fields (and rings) in a rather natural way. From the very beginning of the project, OMEGA was in fact designed to have an optimized internal module to support these services, since they were one of the prerequisite basis for the coding theory computations that we had in mind. We soon found out, however, that we could also achieve a general system for mathematical computations, and the present OMEGA is the result of having worked as far as possible this possibility.

The fields $\mathbb{Z}_p$

The most basic constructor is the function `Zn p`, which creates, if p is a positive integer, the ring $\mathbb{Z}_p$ of integers mod p (and so the field $\mathbb{Z}_p$ if p is prime). For example, 1234567891 is a prime number, and so

```
p=1234567891
K = Zn p
---> Z(1234567891) :: Field
```

creates the field, which we call K, of p elements.

The order of an element $x \uparrow 0$ of a finite field (for example $x \in K$) is obtained with the function `ord(x)`. As we know, it must be a divisor of p-1. Actually, if d is a divisor of p-1 then there are exactly $\phi(d)$ elements of order d in K. In particular, there are $\phi(p-1)$ elements of order p-1, that is, generators of the multiplicative group of non-zero elements (such elements are called *primitive* elements of the field).

We can find the list of divisors of p-1:

```
d=divisors(p-1)
--->
{1, 2, 3, 6, 9, 18, 5, 10, 15, 30, 45, 90,
 3607, 7214, 10821, 21642, 32463, 64926,
 18035, 36070, 54105, 108210, 162315, 324630,
 3803, 7606, 11409, 22818, 34227, 68454,
 19015, 38030, 57045, 114090, 171135, 342270,
 13717421, 27434842, 41152263, 82304526,
 123456789, 246913578, 68587105, 137174210,
 205761315, 411522630, 617283945, 1234567890}
 :: List
```

and the corresponding values of the Euler's ϕ function:

```
phi={phieuler (d.i) with i in range(d)}
--->
{1, 1, 2, 2, 6, 6, 4, 4, 8, 8, 24, 24, 3606,
 3606, 7212, 7212, 21636, 21636, 14424, 14424,
 28848, 28848, 86544, 86544, 3802, 3802, 7604,
 7604, 22812, 22812, 15208, 15208, 30416,
 30416, 91248, 91248, 13710012, 13710012,
 27420024, 27420024, 82260072, 82260072,
 54840048, 54840048, 109680096, 109680096,
 329040288, 329040288} :: List
```

Let us check that the sum of the $\phi(d)$ is equal to p-1, as it should be:

```
sigma phi.i with i in range(phi)
---> 1234567890 :: Z
```

Here is a simple experiment with orders: we take ten random nonzero elements of K and for each one we also find its order and the quotient of p-1 by it:

```
{{x=random(1,p-1):K, ord(x),(p-1)/ord(x)} with
  i in 1..10}
--->
{{802534091, 246913578, 5}, {653136751,
 1234567890, 1}, {56958461, 1234567890, 1},
 {455711964, 617283945, 2}, {1100421531,
 246913578, 5}, {1176514037, 617283945, 2},
```

```
{961302966, 1234567890, 1}, {1038759249,
41152263, 30}, {382461739, 617283945, 2},
{804459382, 411522630, 3}} :: List
```

The expression x:K is nothing but the class of the integer x mod p (if x is a value and T is a type, the value of x:T is the value of x when regarded as type T, provided the conversion from the natural type of x to the type T makes sense). Note that in the output above the quotient (p-1)/ord(x) is small. In fact, it is easy to find the list of pairs q,p formed with a given possible quotient and its likelihood to appear. The first few terms of this list are

```
{{ 1, 0.26652}, { 2, 0.26652}, { 3, 0.08884},
 { 6, 0.08884}, { 9, 0.04442}, {18, 0.04442},
 { 5, 0.06663}, {10, 0.06663}, {15, 0.02221},
 {30, 0.02221}, {45, 0.01111}, {90, 0.01111}}
```

(the probabilities of the remaining quotients are less than 10^{-4}, and most of them are much smaller). Thus, 1 or 2 should appear about 52%, 3 or 6 about 17%, 5 or 10 about 13%, 9 or 18 about 9%, and 15 or 30 about 4%.

In the experiment above we see, for example, that

```
x=653136751:K
```

is a primitive element of K. Hence (note that 3607 is a divisor of p-1)

```
y=x^((p-1)/3607)
---> 1118708644 :: K
```

will be an element of order 3607:

```
ord(y)
---> 3607 :: Z
```

As already mentioned, prim(K) returns a primitive element of K:

```
prim(Zp)
---> 3 :: Z(1234567891)

z=(3:K)^((p-1)/3607)
---> 1058784230 :: K

ord(z)
---> 3607 :: Z
```

Construction of extensions

The other main function to construct finite fields is ext. If k is a finite field, f = f(T) ∈ k[T] is a monic irreducible polynomial in the indeterminate T and t is another indeterminate, then

```
ext(k,t,f)
```

constructs the field F = k[T]/(f). After the function call, t is bound to the class of T mod f, so that the elements of F have the form $\alpha_0+\alpha_1 t+\ldots+\alpha_{r-1}t^{r-1}$, where r is the degree of f and $\alpha_i \in k$. The same result can be obtained by the function call

```
ext(k,f(t))
```

In order to make the function ext practical, we need a way to find irreducible polynomials over k. This is accomplished by the function polirred. More specifically,

```
polirred(k,r,T)
```

yields a monic polynomial of degree r with coefficients in k in the indeterminate T and which is irreducible over k. For example, if k=Zn 17, the following call produces a list of 16 monic irreducible polynomials over k with successive degrees in the range 2..17:

```
{polirred(k,i,T) with i in 2..17}
--->
{T^2+14, T^3+T+3, T^4+14, T^5+T+3, T^6+T+7,
 T^7+T+5, T^8+14, T^9+T+3, T^10+T+7,
 T^11+3T+7, T^12+T+2, T^13+2T+6, T^14+T+8,
 T^15+3T+6, T^16+14, T^17+16T+16} :: List
```

Therefore, we can construct the field of 17^{17} elements as follows:

```
F=ext(k,t^17-t-1)
---> F(17^17) :: Field
```

And now we can operate in F. For example:

```
n=card(F)-1
---> 827240261886336764176 :: Z

r=ord(t)
---> 51702516367896047761 :: Z

n/r
---> 16 :: Z

minpol(t,k,T)
---> T^17+16T+16 :: Z17[T]

minpol(t^700,k,T)
---> T^17+7T^15+16T^14+12T^12+13T^11+7T^10+
     3T^9+16T^8+2T^7+4T^6+8T^5+12T^4+12T^3+8T
     ^2+16 :: Z17[T]
```

If needed, OMEGA can produce the list of monic irreducible polynomials over a finite field K, of a given degree r. The function is

```
listirred(K,r,T)
```

where T is the indeterminate in which we want to write the polynomials.

Two more internal functions worth being mentioned are

```
factor(f,K)
irred(f,K)
```

The first factors the polynomial f with coefficients in the finite field K into its irreducible factors, and the second verifies whether the polynomial f with coefficients in K is irreducible over K.

Relations

Relations are constructs of the form

```
{a->x,b->y,...}
```

If we assign this object to a variable t, then t(a) returns x, t(b) returns y, and so on. We see that a,b,... bevave as *indices*, or *keys*, and x,y,... as corresponding *associated values*.

In the following table, the indices are the numbers n less than 27000 that are the sum of two cubes, and the corresponding values are pairs (i,j) such that $i \leqslant j$ and $n = i^3+j^3$:

```
L={i^3+j^3 -> {i,j} with (i,j) in (1..30)^2
   where i<=j}
```

The result is a table of dimension 465 and so we have omitted it. Instead, we have selected the elements of the table that have more than one decomposition:

```
select(L,(x,y)—> nops(L(x))>1)
---> {1729->({1, 12},{9, 10}),
      4104->({2, 16},{9, 15}),
      13832->({2, 24},{18,20}),
      20683->({10, 27},{19, 24})} :: Table
```

Thus, 1729 is the first integer that is the sum of two cubes in two different ways, and we also get the next three integers with the same property.

Remark. Note that

```
{1->a,2->b,1->c}
---> {1->(a,c),2->b}
```

and so the values associated to the same key are accumulated into a sequence of values for that key. On the other hand, the construction of L could be as follows:

```
L={i^3+j^3 -> {i,j} with i,j in 1..30,1..30
   suchthat i<=j}
```

In other words, the parenthesis in `(i,j)` are unnecessary (with or without parenthesis, we are dealing with a sequence of two elements), and for a range r, r^2 is equivalent to `(r,r)`, or just `r,r`.

2. Block error-correcting codes

In this section, we introduce the most basic concepts of the theory of error-correcting codes, and, where appropriate, we mention or add the related OMEGA functions.

Hamming distance. Weight

Let T be a finite set and let us consider the set T^n. We can think of the elements of T^n as *words* of length n formed with the symbols of the *alphabet* T.

Whenever T is a finite field F, instead of words of length n we will also say *vectors* of dimension n, inasmuch as F^n is a vector space of dimension n over F.

If $x \in T^n$, the *support* of x, *support*(x), is the set of the indices $i \in \{1,\dots,n\}$ such that $x_i \uparrow 0$.

We set

$$\mathit{weight}(x) = |x| = |\mathit{support}(x)|$$

(*weight*, or *Hamming norm*, of x).

Finally, if $x,y \in T^n$, we write

$$d(x,y) = |x-y|,$$

that is, the number of indices i such that $x_i \uparrow y_i$, and we say that it is the *Hamming distance* between x and y.

It is easy to write Ω-functions, which we will call `support`, `weight` and `dist`, that implement the functions above.

```
support(x) := {i where x.i != 0 with i in
                range(x)}
weight(x)  := dim(support(x))
dist(x,y)  := weight(x-y)
```

In the following examples, we use the function `rv(n,m)` introduced in section 1.4:

```
Examples:
x=rv(8,2)
---> [0, 1, 1, 0, 0, 0, 1, 0] :: Vector(Z)
y=rv(8,2)
---> [0, 0, 1, 1, 1, 1, 1, 0] :: Vector(Z)
weight x
---> 3 :: Z
weight y
---> 5 :: Z
dist(x,y)
---> 4 :: Z
```

The purpose of the theory of error-correcting codes

In the theory of error-correcting block codes, the following aspects are considered:

- First, we group the symbols (of the alphabet T) of an information stream into words (or *blocks*) u of some length k (thus $u \in T^k$).
- Second, each u is *coded* into a word $x \in T^n$, for some $n > k$, in a way that the map

$$\alpha: u \mapsto x$$

- (called *coding function*) is one-to-one. Let us write $C \subset T^n$ to denote the image of α.
- Third, we "transmit" x through a "channel", possibly with "noise", and at the end we "receive" a word $y \in T^n$.
- Then, the most important step, the *decoding* process, is done by means of a map β (called *decoding function*) of a subset $D \subseteq T^n$ onto C,

$$\beta: y \mapsto x'.$$

  If $y \notin D$, y is considered *non-decodable*. When $D = T^n$, we say that the decoding is *complete*.
- Finally, we take the element $u' \in T^k$ such that $\alpha(u') = x'$ as the decoded block.

Here is now the key definition: If t is a positive integer, we say that β has *correcting capability* t if $y \in D$ and $x' = x$ (hence also $u' = u$) whenever $d(x,y) \leqslant t$. In other words, if

not more than t symbols of x have been altered along the transmission channel, then we have that y is decodable and that the decoded word x′ coincides with the transmitted vector x.

It can be seen, under quite general circumstances (cf. [17], Theorem 4.2.3, p. 132), that the maximum possible correcting capability of a code of length n is $\lfloor (d-1)/2 \rfloor$, where d is the minimum of the numbers d(x,y) for $x,y \in C$, $x \uparrow y$ (the integer d is called the *minimum distance* of the code).

The integers n and k are the *length* and the *dimension* of the code, respectively. The quotient k/n is called the *transmission rate*. We say that a code is of type [n,k,d] if its length is n, its dimension k and its minimum distance d.

The goal of the theory of error-correcting block codes is to find codes with high transmission rate and high correcting capability. The two parameters, however, cannot be improved independently. Indeed, if d is the minimum distance (and so $t = \lfloor (d-1)/2 \rfloor$ is an upper bound for the correcting capability), then $d+k \leqslant n+1$ (Singleton bound; see Theorem 4.5.6, p. 174, in [17]).

The repetition code [3,1,3] is a simple example that illustrates the notion of effective coding and decoding. Note that the operator |, which has been used in section 1.2 to join vectors and matrices, can also be used as the or boolean operator.

```
# Repetition code Rep3 = [3,1,3]

# Encoder
a(u) := [u,u,u]   # transmission rate = 1/3,
                  # minimum distance = 3

# Decoder by «majority», if majority exists;
# corrects one error
b(y) :=
  if y.1 == y.2 | y.1 == y.3 then y.1
  elif y.2 == y.3 then y.2
  else print(«Non-decodable vector»)
  end

# Let us make a list of the length 3 binary
# vectors and the corresponding decoding
# values:
{[i,j,k], b([i,j,k]) with (i,j,k) in (0..1)^3}
--->
{[0, 0, 0],0,  [0, 0, 1],0, [0, 1, 0],0,
 [0, 1, 1],1,  [1, 0, 0],0,  [1, 0, 1],1,
 [1, 1, 0],1, [1, 1, 1],1} :: List
```

Codes defined by a control matrix

A *control matrix* of codimension r for F-vectors of length n is a matrix $H \in M_r^n(F)$ of rank r.

The *syndrome* of a vector $y \in F^n$ is

$$s = yH \in F^r.$$

The vectors whose syndrome is 0 form a subspace C of dimension k = n-r of F^n and we say that C is the *code defined by* H.

If $G \in M_n^k(F)$ is a matrix whose rows $G^1,\ldots,G^k$ form a linear basis of C over F, we say that G is a *generating matrix* of C. In this case, we can take the injective linear map $F^k \to F^n$ such that

$$u \mapsto x = uG$$

as a coding function. Since

$$uG = u_1G^1+\ldots+u_kG^k,$$

it is clear that this map sets up an isomorphism of F^k onto C.

To give a first example of these concepts, we list an Ω-library that defines the (binary) Hamming [7,4,3] code and the corresponding coding and decoding functions (cf. [17], p. 255).

```
# The binary Hamming code C=[7,4,3]

# let K be the field of binary digits
let K=Zn 2

# let u be the unit of K
u=1:K

# Let R be the matrix whose columns are the
# binary vectors of length three and weight
# at least 2
R=[[u, 0, 1, 1],
   [1, 1, 0, 1],
   [0, 1, 1, 1]]
# (since one entry is in K, all other entries
# are projected to K)

# Generator matrix of C: the transpose of
# stacking R on top of the identity matrix I_4
G = (R & Id(4))'

# The control matrix of C: stack the identity
# matrix I_3 on top of the transpose of R
H = Id(3) & R'

# Encoder:
hamming_encoder(u) := u*G
# (note that u*G = uR'|u)

# Decoder
hamming_decoder(y):=
  begin local n,r,s,j,e,x
    n=dim(H); r=dim(G)   # n=7, r=4 (number of
    rows of H and G)
    s=y*H                # syndrome of y
    if zero?(s) then
       show(«0 errors»)
       x=y
    elif
      j=index(H,s)  # position of s in H
      show(«Error in position »,j)
```

```
      e=eps(j,n)    # error vector
      x=y-e         # correcting the error
    end
    take(x,-r)      # return the last 4
                      components of x
  end;

# Example
u=[1,1,0,1]

x=hamming_encoder(u)
---> [0,1,0,1,1,0,1] :: Vector(K)

# let us simulate an error in position 4
e=eps(4,7)
---> [0,0,0,1,0,0,0] :: Vector(Z)
y=x+e
---> [0,1,0,0,1,0,1] :: Vector(K)

hamming_decoder(y)
---> Error in position 4
[1,1,0,1] :: Vector(K)

hamming_decoder(x)
---> 0 errors
[1,1,0,1] :: Vector(K)
```

Let us see briefly why the Hamming decoder works. We have that s = yH = (x+e)H = eH, since x is a code-word, and that eH is (in the example) the *fourth* row of H, so the position index of s in H indicates the position of the error.

3. Alternant codes

In this section, we will study the class of alternant codes, and some relevant subclasses, and we will show how to construct them using OMEGA. For the mathematical part of the alternant codes, we will follow very closely [17], §8.3.

The *control matrix* of an *alternant code* of length n and order r has the following form:

$$H = \begin{pmatrix} h_1 & \cdots & h_n \\ h_1\alpha_1 & \cdots & h_n\alpha_n \\ \vdots & & \vdots \\ h_1\alpha_1^{r-1} & \cdots & h_n\alpha_n^{r-1} \end{pmatrix}$$

Here, $h_1,\ldots,h_n$ and $\alpha_1,\ldots,\alpha_n$ are elements of some finite field K′, with $h_i, \alpha_i \uparrow 0$ for all i and $\alpha_i \uparrow \alpha_j$ for all $i \uparrow j$. We will refer to $h = [h_1,\ldots,h_n]$ and $\alpha = [\alpha_1,\ldots,\alpha_n]$ as the *vector* h and the *vector* α of the control matrix. Here is an Ω-function that constructs H:

```
ACM(h,a,r):=
begin local H, # the matrix
            f # current vector of H
  f=h; H=[f]
  for i in 1..(r-1) do
     f=prod(f,a); H = H & f
  end
  H'
end
#
ACM(a,r):= ACM(a,a,r)
```

The function prod(a,b) of two vectors a and b of the same length, which returns their component-wise product, can be defined as follows:

```
prod(a,b):= [a.i*b.i for i in range(a)]
```

If K is a subfield of K′ (possibly K = K′), then the code defined by H over K is the subspace of K^n whose elements are the vectors x such that xH = 0 (vectors with null syndrome).

To implement the alternant codes, we need to store the vectors h and a, and the codimension r. Since on decoding we will have to calculate syndromes, it is also advisable to calculate and store H. We will also extensively use the inverses of the elements of α, so it is convenient to have precomputed the vector β of these inverses. The more convenient way for us to achieve this here is to assign the five objects h,α,β,H,r to the global variables vh, va, vb, H, cd, respectively. This way, we will ensure that at decoding time we will have to carry out operations that involve only the vector y and precomputed data about the code (without the precomputations, they would have to be repeated for each vector y, and so the decoder would be inefficient).

```
# To construct alternant codes
Alternant(h,a,r):=
begin
  vh=h; va=a; vb=inv(a); H=ACM(h,a,r); cd=r
end
#
Alternant(a,r):= Alternant(a,a,r)
```

This listing illustrates the function overloading capacity of OMEGA, as the call `Alternant(a,r)` is defined in terms of the more general call `Alternant(h,a,r)`. So we can use the same name for a function of 3 or 2 parameters. Using types the overloading can also be used for the same number of parameters. For example,

```
f(x:Rational):=1
f(x):=0
```

defines a function f that on rational numbers takes the value 1 and otherwise the value 0.

Note that the function ACM can be used to define the Vandermonde matrices.

```
# Vandermonde matrix of codimension r on the
# vector a
vandermonde(a,r) :=
  ACM(constantvector(1,dim(a)),a,r)'

# Vandermonde square matrix on the vector a
vandermonde(a):=
  ACM(constantvector(1,dim(a)),a,dim(a))'
```

BCH codes

These codes can be treated as special alternant codes. A BCH code over a finite field K (cf. [17], §8.1) depends on an element a in a finite field extension K′/K, a positive integer d, called the *designed distance*, and a non-negative integer l, and it is not difficult to see that it coincides, if n is the order of a, with the code defined over K by the alternant control matrix of order r = d-1 corresponding to the vectors h = $[1,a^l,a^{2l}\ldots,\alpha^{(n-1)l}]$ and $\alpha = [1,a,a^2,\ldots,a^{n-1}]$. The BCH codes in the *strict*, or *narrow*, sense are those with l = 1.

To make explicit how to use OMEGA to define BCH codes, it is convenient to define an auxiliary function series(a,n) that returns $[1,a,\ldots,a^n]$:

```
series(a,n) := [a^i with i in 0..(n-1)]
```

Then we can define the BCH code associated to a, d and l as follows:

```
# BCH codes, general and in the narrow sense
BCH(a,d,l):=
  begin local n=ord(a)
     Alternant(series(a^l,n),series(a,n),d-1)
end
#
BCH(a,d):= BCH(a,d,1)
```

RS codes

Next we show how to construct RS codes (cf. [17], §8.2). They can be seen as a special case of BCH codes in the strict sense, obtained when K′ = K, a is a primitive element of K and d = r+1, r being a positive integer (r < n) called the *codimension* of the RS code. Thus, a RS code of codimension r is associated to a given finite field K and can be defined as follows:

```
# RS code of codim r associated to the field K
RS(K,r) := BCH(prim(K),r+1)
```

Classical Goppa codes

Another class that is a special case of the alternant codes are the classical Goppa codes. A *classical Goppa code* over K is associated to a vector α of length n with components in the field K′ (a finite extension of K) and a univariate polynomial g of degree r with coefficients in K′ such that $g(\alpha_i) \uparrow 0$ for all i, and can be defined (see [17], p. 389-395) as the alternant code with vector h equal to the inverses of the values of g on the components of α, vector α equal to α, and with order r equal to the degree of g. So OMEGA can be used as follows:

```
# Classical Goppa codes
Goppa(g,a) :=
  Alternant(inv(eval(g,a)),a,deg(g))
```

4. The BM decoder and its Omega implementation

One of the outstanding features of the alternant codes is that there are good decoding algorithms for them. Here, we describe a slightly improved version of the algorithm presented in [17] (Theorem 8.3.8, p. 403), called the Berlekamp-Massey algorithm (cf. [23], §1.6, p. 12-15).

As explained in chapter 3, $\beta = [\beta_1,\ldots,\beta_n]$ is the vector formed with the inverses of the elements α_i, that is, $\beta_i = \alpha_i^{-1}$.

The BM decoding algorithm

This algorithm takes as input a vector $y \in K^n$ and it outputs, if y turns out to be decodable, a code vector x.

1. Find the syndrome s = yH, say $s = [s_0,\ldots,s_{r-1}]$.
2. Transform s into a polynomial S in the indeterminate z:

   $$S = s_0+s_1z+\ldots+s_{r-1}z^{r-1}$$

   (S is called *polynomial syndrome*).
3. Perform the Euclidean algorithm with $r_0 = z^r$ and $r_1 = S$. This means that we find $q_1,q_2,\ldots$ and $r_2, r_3,\ldots$ so that q_j and r_{j+1} are the quotient and the remainder of the whole division of r_{j-1} by r_j, which means that

   $$r_{j-1} = q_j\, r_j + r_{j+1},$$

   with the condition that r_{j+1} has, if non-zero, lower degree than r_j. The process stops when we find a j, say j = k, such that r_k has degree $\geqslant$ r/2 and r_{k+1} has degree < r/2 (in the ordinary Euclidean algorithm, the process stops when $r_{j+1} = 0$). In addition, we also compute polynomials

   $$v_0 = 0, v_1 = 1, v_2, \ldots, v_k$$

   such that

   $$v_{j+1} = v_{j-1}-q_jv_j$$

   (note that $r_{j+1} = r_{j-1}-q_jr_j$). We set $\sigma = v_k$ (this is called the *error locating polynomial*) and $\varepsilon = r_k$ (this is called *error resolving polynomial*).
4. Make a list $L = \{m_1,\ldots,m_s\}$ of the indices $j \in L = \{1,\ldots,n\}$ such that $\sigma(\beta_j) = 0$. These indices are called *error locations*. If s is less than the degree of σ, return the error message "Non-decodable vector''.
5. If $K = Z_2$, return the result of replacing 0 by 1 and 1 by 0 in y for all the error locations.
6. Otherwise, for each $j \in L$, replace the value y_j by

   $$y_j + \frac{\alpha_j \cdot \varepsilon(\beta_j)}{\sigma'(\beta_j)},$$

   where σ' is the derivative of σ.

Theorem. *The Berlekamp-Massey algorithm corrects at least r/2 errors.*

• *Proof*. See [17], Theorem 8.3.8, p. 403.

Ω-implementation

First, we describe two Ω-functions that are basic engines of the BM algorithm: a modification of the euclidean algorithm, as needed in step 3 of the algorithm, and whose purpose is to solve the so-called key equation, and a search function that supplies the error positions.

```
# Modified Euclidean Algorithm, solver of the
# key equation
euclid_bm(r0,r1,t):=
begin
  local r, v0,v1,v, q
  v0=0; v1=1
  while t <= deg(r1) do
     q=quo(r0,r1)
     r=rem(r0,r1); r0=r1; r1=r
     v=v0-q*v1;    v0=v1; v1=v
  end
  {v1,r1}
end
```

The returned pair contains the polynomials `v1` and `r1`, which, in the case of the BM algorithm, are the error-locator polynomial and the error-evaluator polynomial, respectively.

The other basic function lists, given a polynomial f and a list or vector a = [a_1,…,a_n], the indices j in {1,…,n} such that a_j is a zero of f.

```
zero_positions(f,a):= {j suchthat
  eval(f,a.j)==0 with j in range(a)}
```

Now we can deal with the BM decoder. We assume that we have the global variables vh, va and vb, holding vectors of the same length n, with vb the component-wise inverse of va, and the global variables `H` and `r`, holding the control matrix and the rank of the alternant code (see section 3). In the definition of `H`, a base field K and an extension K′/K will also be defined, and in the listing below we will only assume that the base field is held in the variable K. As it can be noted, the implementation provided by this listing is just a straightforward translation of the mathematical algorithm.

```
bm(y):=
begin
  local  s,    # syndrome
         S,    # polynomial syndrome
         key,  # solution of the key equation
         lp,   # error-locating polynomial
               # (sigma)
         ep,   # error-evaluating polynomial
               # (epsilon)
         L,    # list of error positions
         j,    # a component of L
         E,    # error list corresponding to L
         e     # a component of E
  s=y*H
  if zero?(s) then return y end
  clear z
  S=vector2pol(s,z)
  key=euclid_bm(z^cd,S,cd//2)
  lp=key.1; ep=key.2
  L=zero_positions(lp,vb)
  if dim(L)<deg(lp) then
    return(«Non decodable vector») end
  if K==Zn 2 then
    show(«Error positions: »,L)
    return flip(y,L) end
  lp=der(lp);  # only the derivative of lp is
               # needed below
  E=null   # start with the null sequence
  for i in range(L) do
    j = L.i
    e = va.j * eval(ep,vb.j) /
        (vh.j * eval(lp,vb.j))
    E=(E,-e)
    y.j = y.j + e
  end
  show(«Error table: »,{L,{E}})
  y
end
```

Remark

We have used the function `flip(y,L)`, whose action is to replace 0 by 1 and 1 by 0, in the vector y, for the positions indicated in the list L: `flip (y,L):=for j in L do y.j=1-y.j end`

5. Examples of BM decoding of RS codes

In this section, we have collected examples of the working of the BM algorithm, for a few chosen RS codes, that can be run in OMEGA. The idea is to illustrate the complexity of decoding when both the code rate and the correcting capacity are gradually increased.

We have tried to write functions that are useful not only for RS codes, but for other codes as well, and to gather them meaningfully. Because of the lack of space, examples of other classes of codes have been ruled out, but can be found elsewhere ([27], [26]).

The RS codes have the property that k+d = n+1 = q, where k and d are the dimension and the minimum distance, respectively, and q is the cardinal of the base field K. In other words, the Singleton inequality is an equality for these codes, a fact that usually is expressed by saying that the code is *maximum distance separable*, or MDS for short. In fact, k = n-r, because the control matrix has rank r, and it is a general fact for BCH codes that d ⩾ r+1 (the so-called BCH bound), which, together with the Singleton bound, imply that d = r+1 = (n-k)+1.

Now suppose we want an RS code with a rate of at least ρ that can correct t errors. What is the minimum possible q we have to take? The answer is easy, as we are assuming that k ⩾ ρn and n = q-1, so that q = n+1 = k+d ⩾ ρq-ρ+2t+1, or

$$(1-\rho)q \geqslant 2t+(1-\rho).$$

If we set r = (1-ρ), which we will call the *redundancy rate* of the code, then the condition is

$$q \geqslant 1+2t/r.$$

Let us write an Ω-function that finds, for given r and t, the minimum q that satisfies the equality.

```
# To find the least number q which is a prime
# power that is greater or equal to a given
# positive number x
next_q(x):=
  begin local q
    q=ceil(x)
    while primepower?(q)==false do
      q=q+1
    end
    q
  end
#
# To find the least number q that is a prime
# power which is equal or greater than
# 1+2t/r, t and r given
next_q(r,t) := next_q(1+2*t/r)
```

For r in {0.40,0.35,0.30,0.25,0.20} (corresponding, respectively, to the transmission rates {0.60,0.65,0.70, 0.75,0.80}), and for t in {1,2,...,12}, we can compile a table of the minimum q required:

```
R=[0.40,0.35,0.30,0.25,0.20]
T=[1..12]
[[next_q(r,t) with t in T] with r in R]
```

ρ\t	1	2	3	4	5	6	7	8	9	10	11	12
0.60	7	11	16	23	27	31	37	41	47	53	59	61
0.65	7	13	19	25	31	37	41	47	53	59	64	71
0.70	8	16	23	29	37	41	49	59	61	71	79	81
0.75	9	17	25	37	41	49	59	67	73	81	89	97
0.80	11	23	31	41	53	61	71	81	97	101	113	121

Example: RS[26,16,11]
Suppose we want an RS code with a rate of at least 60% that corrects at least 5 errors. The table above tells us that the minimum q is 27. So let us set q = 27, hence n = 26. Since t = 5, the least codimension is r = 10 and so k = 16.

The construction of this code is now very straightforward:

```
Z3=Zn 3
clear x
f=polirred(Z3,3,x)
K=ext(Z3,f)
n=card(K)-1; r=10
RS(K,r)
```

Now we would like to run the BM decoder to see whether it works as expected. However, before we will provide some tools for creating arbitrary random error patterns and code vectors.

Random combinations of m elements chosen among n elements. The following function delivers a random combination of m distinct elements in {1,2,…,n}:

```
rd_comb(n,m) :=
  begin local c,x
    c={null}
    while dim(c)<m do
      x=random(1,n)
      if index(c,x)==0 then c=c|{x} end
    end
  end
```

Random lists of elements of K. First let us produce a random non-zero element of a finite field K, by composing the function `elem(j,K)` that produces the j-th element of K (with some natural built-in order for the elements of K) and the function call `random(1,q-1)`, which produces a random integer in the interval [1,q-1]:

```
rd(K) := elem(random(1,q-1),K)
```

If we want a list of s non-zero random elements of K, we can call the function

```
rd(s,K) := {rd(K) with i in 1..s}
```

Random error patterns. Given the length of the code, n, a weight s, $0 \leqslant s \leqslant n$, and the field K, we can produce a random error pattern of length n, weight s, with entries in the field K, with the following function:

```
rd_err_vec(s,K,n) :=
  begin local E, L, e, i
    E=rd(s,K); L=rd_comb(n,s)
    e=constantvector(0,n)
    for i in range(E) do
      e=e+(E.i)*eps(L.i,n)
    end
  end
```

If we do not specify n, let its default be n = *card*(K)-1:

```
rd_err_vec(s,K) := err_vec(s,K,card(K)-1)
```

Finding code vectors. If a is the primitive element of K with which we constructed the control matrix H, then h = [1,a,a^2,…,a^{n-1}] and α = h, and so the columns of H have the form [1,a^i,a^{2i},…,$a^{(n-1)i}$], i = 1, …, r (the Vandermonde matrix of n rows on the elements a,a^2,…,a^r). It follows from this, and the fact that the sum of the `series(b,n)` is zero if b is an element of K-{0,1}, that the vectors x_j of the form x_j = [1,a^j,a^{2j},…,$a^{(n-1)j}$] are code vectors for j = 0,…, k-1. Actually the vectors x_0,…,x_{k-1} are linearly independent (note that the corresponding matrix is the Vandermonde matrix of k rows on 1,a,a^2,…,a^{n-1}) and so form a basis of the code. Here is a direct construction of this generator matrix:

```
rs_G(a,n,k) :=
    [series(a^j,n) with j in 0..(k-1)]
```

A final remark is that if we want a random code vector, we have to produce a random linear combination of the rows of G. This can be done with the following function:

```
rd_lin_comb(G) :=
  begin k,n, t
    k=dim(G); n=dim(G.1)
    t=[elem(random(n+1),K) with i in 1..k]
    t*G
  end
```

Decoding trials. To simulate the coding + transmission (with noise) + decoding situation, we can set up a function that successively generates a random code vector x, adds to it a random error-pattern e of a prescribed number of errors, and calls the BM decoder on y = x+e. To better track the results, we will, before calling BM, print x and a pair consisting of the support of e and the vector of non-zero entries of e (this pair is the real error pattern). Since BM prints the error locations and the error list, the operation will be fine if these coincide with the error-pattern printed before. A final check is that instead of returning `bm(y)`, we will return `x-bm(y)`, which should be zero if s is not greater than the error-correcting capacity r/2. Here is the corresponding Ω-function:

```
rs_decoder_trial(s):=
  begin local x, e
    x=rd_lin_comb(G,K)
    show(«Random code vector:», x)
    e=rd_err_vec(s,K)
    show(«Error pattern of trial:»,
      {support(e),non_zeros(e)})
    x-bm(x+e)
  end
```

Now it will be enough to evaluate expressions of the form rs_decoder_trial(s) for diverse s. Let us list the final file containing the complete example:

```
# RS[26,16,11]. Corrects 5 errors

Z3=Zn 3
clear t
K=ext(Z3,t^3+2*t+1)

n=card(K)-1; r=10

RS(K,r)

G=rs_G(t,n,n-r)

rs_decoder_trial(3)
rs_decoder_trial(5)
rs_decoder_trial(6)  # beyond the correcting
                     # capacity
```

Let us also list the output for a trial with 5 errors:

```
rs_decoder_trial(5)
---> Random code vector:
[1, t^2+2t, t^2+2t+1, t, 2t^2+t, t^2+t, t^2+t,
 t, 2t^2+2t+2, t^2+2t+1, t^2, t+1, t^2+t+2,
 2t+2, 2t^2+2t+2, t^2+2t, 0, t, 2t^2+2, 1,
 2t^2+2, 2t^2+2t+1, t+1, 2t^2+t, 2t^2+1, 2t+1]

Error pattern of trial:
{{2, 6, 8, 16, 26}, {2t^2+2t, 2t^2+2, t^2+2,
  t^2+t+1, t^2+2t+2}}

Error table:
{{2, 6, 8, 16, 26}, {2t^2+2t, 2t^2+2, t^2+2,
  t^2+t+1, t^2+2t+2}}
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
:: Vector(Z3)
```

Example: RS[36,24,13]
Let us work out the implementation of an RS code with rate of at least 65% and correcting 6 errors or more. The minimum q is 37. Thus, we set q = 37, hence n = 36. Since t = 6, the least codimension is r = 12 and so k = 24 (k/n is indeed greater than 0.65). Now we can proceed as in the previous example, and it will suffice to list the OMEGA file.

```
K=Zn 37
n=card(K)-1; r=12

RS(K,r)
t=prim(K)

G=rs_G(t,n,n-r)

rs_decoder_trial(4)
rs_decoder_trial(6)
rs_decoder_trial(7)  # beyond correcting
                     # capacity
```

Example: RS[48,34,15]
For an RS code with a rate not less than 70% and correcting at least 7 errors, the least possible q is 49. Hence n = 48. Since t = 7, the least codimension is r = 14 and so k = 34 (k/n is indeed greater than 0.70). Here is a listing of the corresponding OMEGA file.

```
Z7=Zn 7
K=ext(Z7,t^2-2*t-2)  # t coincides with
                     # prim(K)
n=card(K)-1; r=14

RS(K,r)

G=rs_G(t,n,n-r)

rs_decoder_trial(5)
rs_decoder_trial(7)
rs_decoder_trial(8) # beyond correcting
                    # capacity
```

Example: RS[80,60,21]
For an RS code with a rate not less than 75%, and correcting at least 10 errors, the least q is 81, and n = 80. Since t = 10, the least codimension is r = 20 and so k = 60 (k/n is exactly 0.75). Here is a listing of the corresponding OMEGA file.

```
Z3=Zn 3
K=ext(Z3,t^4+t+2)  # t is primitive; found
                   # with polirred(Z3,4,t)
n=card(K)-1; r=20

RS(K,r)

G=rs_G(t,n,n-r)

rs_decoder_trial(7)
```

```
rs_decoder_trial(10)
rs_decoder_trial(11)  # beyond correcting
                      # capacity
```

If, instead of q = 81, we take q = 83, and still r = 20, we get an RS[82,62,21] that performs very similarly, with a slightly better rate, but with the faster arithmetic mod 83 than that of F_{81}.

Example: RS[120,96,25]
The least q for an RS code with a rate not less than 80% and correcting at least 12 errors is q = 121, hence n = 120. The least codimension is r = 24 and so k = 96 (k/n is just 0.80). Here is a listing of the corresponding OMEGA file.

```
Z11=Zn 11
K=ext(Z11,t^2+4*t+2)  # t is primitive
n=card(K)-1; r=24

RS(K,r)

G=rs_G(t,n,n-r)

rs_decoder_trial(9)
rs_decoder_trial(12)
rs_decoder_trial(13)  # beyond correcting
                      # capacity
```

A word on organization
Since running the examples presupposes having loaded the file defining alternant codes, the file implementing the BM algorithm, and a file of tools, all the examples on RS codes have been organized in a file that we list next.

```
# File:   RS.OM
# Needs:  Omega-Athens/99
# Does:   Examples of RS codes and BM decoding
# Author: S. Xambó
# Date:   17/8/99
# Projecte OMEGA / 1999
# S.Xambó, D.Marquès, R.Eixarch, M.Castells,
# D.Arso, P.Garriga

load «e:/omega/xlib»

traces true  # in case we want a step-wise
             # run

# Several examples follow. Uncomment the one
# to be run, and read it at the Omega screen
# by clicking File/Open and choosing rs.omd.
# Alternatively, write the expression read
# '«e:/rs/rs»;' at the Omega prompt and press
# return.

#read «e:/rs/rs[8,4,5]»
#read «e:/rs/rs[12,8,5]»
#read «e:/rs/rs[18,12,7]»
#read «e:/rs/rs[26,16,11]»    # rate=0.60, t=5
#read «e:/rs/rs[36,24,13]»    # rate=0.65, t=6
#read «e:/rs/rs[48,34,15]»    # rate=0.70, t=7
#read «e:/rs/rs[80,60,21]»    # rate=0.75, t=10
#read «e:/rs/rs[82,62,21]»    # rate=0.75, t=10
#read «e:/rs/rs[120,96,25]»   # rate=0.80, t=12

traces false
```

---

6. Some auxiliary and complementary packages

In this chapter we are going to look at several Ω-functions that yield important services in the theory of finite fields and error-correcting codes, and which are interesting on their own from the point of view of computer mathematics.

Some other functions of the tools.om lib
Most of the functions in this library (26 function forms) have already been mentioned and used in these notes, but a few that are very important for the theory of codes have not been considered yet.

The first is a function that yields, given a vector or a list v, a table whose indices are the *different* elements of v and whose associated values are the sequences of positions in which these elements of v occur.

```
distribution(v) := {v.i->i with i in
  range(v)}

distribution({2,3,2,3,2,3,1,2,1,2,3,3,1,3,2})
---> {1->(7,9,14), 2->(1,3,5,8,10,15),
3->(2,4,6,11,12,16)}
```

The following function easily finds, given a vector or list v and an expression e, the list of positions in which e occurs in v.

```
indices(e,v):= {i where e==v.i with i in
  range(v)}
indices(2,{2,3,2,3,2,3,1,2,1,2,3,3,1,3,2});
---> (1,3,5,8,10,15)
```

It is to be remarked that index(e,v) is an internal function that returns 0 if e is not present in v and otherwise the first index i such that $e = v_i$.

To find the values that have maximum frequency in a vector or list, and the corresponding lists of positions that they occupy, we can use the following function:

```
distribution_maximum_frequency(v):=
  begin
    local d, c, m, k, C
    d=distribution(v)
    c={domain(d)}  # the list of keys of
                   # table d
    m=nops(d(c.1)) # number of occurrences
                   # of c.1
    # calculation of the maximum number of
    # occurrences
    for i in 2..dim(c) do
      if (k=nops(d(c.i)))>m then m=k end
    end
```

```
    # Selection of the elements that have
    # maximum frequency
    C=null          # the null sequence
    for i in range(c) do
      if nops(d(c.i))== m then
        C=(C,c.i,{d(c.i)}) end
    end
    C={C}
    {C.(2*i-1)->C.(2*i) with i in
     1..(dim(C)/2)}
  end

distribution_maximum_frequency({2,3,2,3,2,3,1,
  2,1,2,3,3,1,3,2})
---> {2->(1,3,5,8,10,15),3->(2,4,6,11,12,16)}
```

For further useful applications of these functions, in the decoding of algebro-geometric codes, see [26].

The last group of functions in `tools.om` that we want to explain are `brack` and `prima`. Let us consider them in turn. The function `brack(a,x)` (we use a name to remind the way in which it is usually denoted in books on error-correcting codes; see [17], §8.3) yields the list of components of an element a of the finite extension field K[x] of K in terms of the basis $1,x,\ldots,x^{r-1}$ of K[x] over K (so r is the degree of the extension K[x]/K). It uses the internal function `components(a)`, which gives the *sequence* of these components *when a is not in K*.

```
brack(a,x):=
  begin
    if subfield?(field(a),precedent_ext
      (field(x))) then {a}| constantlist(0,
       relativedeg(x))-1)
    else {components(a)}
    end
  end
```

Let us explain a little more the OMEGA services involved in this function. In Ω, a finite field K comes equipped, by its very construction, with a filtration

$$Z_p = K_0 \subset K_1 \subset \cdots \subset K_{n-1} \subset K_n = K$$

such that, for i = 1, …, n, K_i is obtained by a call of the form ext(K_{i-1},f_i), where f_i is an irreducible polynomial over K_{i-1}, and where Z_p has been constructed by a call K_0=Zn p. Given an element x of K, `field(x)` is the subfield K_i such that $x \in K_i$ but $x \notin K_{i-1}$. On the other hand, `precedent_ext(K)` is the field K_{n-1} and `base_ext(K)` is the field K_0. Finally, `subfield?(K,F)` is an external function that tests whether the field K is one of the subfields of the filtration of F:

```
subfield?(K,F):=
  begin local q
    q=card(K)
    if car(K)!=car(F) | q>card(F) then
      return false end
    while card(F)>q do F=precedent_ext(F)
      end
    if F==K then true
    else false end
  end
```

Now the function `prima(h:VEC,x)` takes the vector h of length n over K[x] and makes a vector of length nr over K with the bracks of the components of h, and the function `prima(H:MAT,x)` takes the matrix H of type mxn over K[x] and makes a matrix of type mrxn over K by successively applying `prima` to the rows of H.

```
prima(H:MAT,x):=
  begin local H1, i
    H1=null
    for h in H do H1=(H1,prima(h,x)) end
    [H1]
  end

prima(h:VEC,x):=
  begin local h1, r, v, i
    h1={null}
    for a in h do h1=h1|brack(a,x) end
    [seq h1]
  end
```

For uses of the function prima, see [17], §8.3. To illustrate, let us work out with OMEGA the example 8.3.2 there.

```
load «e:/OMEGA/tools»

Z2=Zn 2

clear x
F=ext(Z2,x^3+x+1)
n=card(F)

f8=[0]|series(x,n-1)  # the set of elements
                      # of F
--->
[0, 1, x, x^2, (x+1), (x^2+x), (x^2+x+1),
 (x^2+1)] :: Vector(F(2^3))

g=T^2+T+1

h=[eval(g,x) with x in f8]
--->
[1, 1, (x^2+x+1), (x+1), (x^2+x+1), (x^2+1),
 (x^2+1), (x+1)] :: Vector(F(2^3))

H= (h & prod(h,f8))'
--->
[1, 0]
[1, 1]
[(x^2+x+1), (x^2+1)]
[(x+1), (x^2+x+1)]
[(x^2+x+1), x]
[(x^2+1), (x+1)]
[(x^2+1), (x^2+x)]
[(x+1), x^2]
:: Matrix(F(2^3))

H1=prima(H,x)
--->
```

```
[1, 0, 0, 0, 0, 0]
[1, 0, 0, 1, 0, 0]
[1, 1, 1, 1, 0, 1]
[1, 1, 0, 1, 1, 1]
[1, 1, 1, 0, 1, 0]
[1, 0, 1, 1, 1, 0]
[1, 0, 1, 0, 1, 1]
[1, 1, 0, 0, 0, 1]
:: Matrix(Z2)

rang(H1)
---> 6 :: Z
```

The volume of Hamming spheres and perfect codes
One remarkable expression in the theory of codes is the volume of the Hamming sphere of radius r in the space F_q^n. This is returned by the function `vol(n,r,q)`, and `vol(n,r)` when q = 2. We can define them by translating to Ω the usual formulae:

```
vol(n,r,q):=
   sigma binomial(n,i)*(q-1)^i with i in 0..r
vol(n,r):=
   sigma binomial(n,i) with i in 0..r

vol(23,3)
---> 2048
vol(11,2,3)
---> 243
```

These functions are used in many others. Here is, for example, an Ω-function to test whether the parameters C = [n,k,d] satisfy the condition for a perfect code:

```
perfect?(C,q):=
        q^(C.1-C.2) == vol(C.1, (C.3-1)//2,q)
perfect?(C):=
        2^(C.1-C.2) == vol(C.1, (C.3-1)//2)

perfect?([23,12,7])   # the Golay binary code
---> true

perfect?([11,6,5],3)  # the Golay ternary
                      # codes
---> true
```

More generally, we can introduce the notion of *perfection* of a code as the quotient of $q^k vol(n,t,q)$, which is the total volume occupied by the Hamming spheres of radius t centered at code vectors, by q^n (the volume of the total space). In this way the perfect codes are those whose perfection is exactly 1, while all others have perfection less than 1 (see next section).

```
perfection(C,q):=
          vol(C.1,(C.3-1)//2,q) / q^(C.1-C.2)
perfection(C):= perfection(C,2)

perfection([11,6,5],3)
---> 1 :: Z
perfection([32,6,15])
---> 0.067
```

Bounds on code parameter's and related functions
There are many bounds other than the Singleton upper bound $k \leqslant n+1-d$, valid under diverse circumstances (cf. [19]), for the dimension of a code.

In this section, we collect a number of Ω-functions for the computation of several bounds of the function $A_q(n,d)$, where $A_q(n,d)$ denotes the maximum possible cardinal for q-ary codes of length n and minimum distance d. For the significance of $A_q(n,d)$, and for the mathematical discussion of several of its known bounds, see [20]). As we will see, the vol function appears in several of the bounds.

Lower bounds
There are only two lower bounds: the Gilbert bound and the Gilbert-Varshamov bound, the latter obtained by reasoning with linear codes. The corresponding Ω functions, `gilbert` and `gilbert_varshamov`, are just a straightforward translation of the usual formulae and procedures:

```
lb_gilbert(n,d,q) := ceil(q^n/vol(n,d-1,q))
lb_gilbert(n,d) := ceil(2^n/vol(n,d-1))

lb_gilbert(10,3)
---> 19

lb_gilbert(11,3)
---> 31

lb_gilbert_varshamov(n,d,q):=
  begin local k
    k=0
    while q^(n-k)>vol(n-1,d-2,q) do k=k+1 end
    k-1
  end
lb_gilbert_varshamov(n,d):=
   lb_gilbert_varshamov(n,d,2)

lb_gilbert_varshamov(10,3)
---> 6
# (compare the value 2^6=64 with
# lb_gilbert(10,3) ---> 19)
```

Upper bounds
The sphere upper-bound for the maximum cardinal of q-ary codes of length n and minimum distance d can be programmed as follows:

```
ub_sphere(n,d,q):=
         floor(q^n/vol(n,(d-1)//2,q))
ub_sphere(n,d):= floor(2^n/vol(n,(d-1)//2))

ub_sphere(10,3)
---> 93
ub_sphere(11,3)
---> 170
```

For binary codes, this bound is improved by the Johnson bound. We introduce the function `ub_johnson`, with two calls, the main one with two parameters and the other, involved in the definition of the first, with three parameters (see [20] for details):

```
ub_johnson(n,d):=
  begin local e,x,i
    if even(d) then n=n-1; d=d-1 end
    e=d//2
    x=(binomial(n,e+1)-binomial(d,e)*
      ub_johnson(n,d,d))/floor(n/(e+1)) + sigma
      binomial(n,i) with i in 0..e
    floor(2^n/x)
  end

ub_johnson(n,d,w):=
  begin local k,b
    k = floor((d+1)/2); b=1
    if k<=w then
      for i in (w-k)..0..(-1) do
        b = floor(b*(n-i)/(w-i))
      end
    else 0
    end
  end

ub_johnson(13,5,5)
---> 23 :: Z
ub_johnson(13,5)
---> 77 :: Z
```

Now we list the Ω code for the Griesmer bound (see [20] for details):

```
ub_griesmer(n,d,q):=
  begin local i
    i=0
    while n>0 do
      n=n-ceil(d/q^i)
      i=i+1
    end
  end
ub_griesmer(n,d):=ub_griesmer(n,d,2)

ub_griesmer(13,5)
---> 6 :: Z
ub_griesmer(14,9,3)
---> 4 :: Z
```

The Ω code for next bound, the Elias bound, contains a local definition of a function, a point that illustrates a powerful feature of OMEGA, since it facilitates to break complicated expressions and procedures into simpler and more meaningful ones. Again see [20] for details about this bound.

```
ub_elias(n,d,q):=
  begin local t, R, f, M, A
    t = 1 - 1/q
    R = 0..(floor(t*n))
    f(r) := r^2 - 2*t*n*r + t*n*d
    M = q^n
    for r in R do
      if f(r)>0 then
        A = floor(t*n*d/f(r) * q^n/vol(n,r,q))
        if A<M then M=A end
      end
    end
    M
  end

ub_elias(n,d):= ub_elias(n,d,2)

ub_elias(14,6)
---> 162
```

Error reduction factor of a code

We code blocks of k information symbols into blocks of n symbols. For a decoder with error-correcting capacity t, let us denote by `err(n,t,p)` the probability that more than t errors occur in a block, where p is the probability that a symbol is altered into another symbol. We can calculate this function by noting that it is equal to 1 minus the probability that at most t errors occur. In other words, $1-\Sigma_{j=0}^{t}\binom{n}{j}p^j(1-p)^{n-j}$. Taking into account the optimization in the calculation of some intermediate expressions, the resulting Ω-function is as follows:

```
err(n,t,p) :=
begin
  locals b,P,q,Q,S
  b=1   # binomial(n,j), for j=0
  P=1   # P = p^j, for j=0
  q=1-p
  Q=q^n # Q=(1-p)^(n-j), for j=0
  S=Q   # The sum S, for j=0
  for j in 1..t do
    b=b*(n-j+1)//j
    P=P*p
    Q=Q/q
    S=S+b*P*Q
  end
  1-S
end
# We set p=0.01 by default
err(n,t):=err(n,t,0.01)
```

Now we would like a function ERF(n,k,t,p) expressing, for a code of length n, dimension d and error-correcting capacity t, and with p as before, the quotient of the average number of erroneous symbols that occur using the code by the that without using the code. The formula for this function is easy to derive and, in Ω terms, is as follows:

```
ERF(n,k,t,p):= k*err(n,t,p)/(n*p)
```

For a code C = [n,k,d], we have $t = \lfloor (d-1)/2 \rfloor$, and we can overload ERF to get the error-reduction factor for C:

```
ERF(C,p) := ERF(C.1, C.2, (C.3-1) // 2, p)
ERF(C):=ERF(C,0.01)

ERF([23,12,7])
---> 0.003968 :: Decimal

ERF([23,12,7],0.001)
---> 0.00000455 :: Decimal
```

Cyclotomic order of an integer q mod n

Often it is required to know the order of an integer q mod n,

provided that gcd(q,n) = 1. Although this is an internal function (which is based on a variation of the algorithm for the one-parameter function ord(a) that gives the order of a non-zero element of a finite field), here is a presentation as an external Ω-function that uses fast modular arithmetic:

```
ord(q,n) :=
  begin local d, # ordered list of divisors of
                 # phi(n)
              A, # integers mod n
              u, # the unit of A
              i, # current index of divisor of
                 # phi(n)
               r # the current divisor of
                 # phi(n)
    if gcd(q,n)>1 then return
      show(«The number »,q,« is not invertible
        mod », n)
    else
      d = set(divisors(phieuler(n)))
      A = Zn n
      q = q:A; u = 1:A
      i=1; r=d.i
      while q^r != u do
        i=i+1; r = d.i
      end
    end
  end
```

For example, the expression

```
n = 571725
{ord(q,n) with q in 2..100 where gcd(q,n)==1}
```

is a list of length 40 that is obtained in 0.7 s.

Cyclotomic classes

In the factorization of X^n–1 over F_q, and assuming gcd(n,q) = 1, the irreducible factors are in one-to-one correspondence with the q-cyclotomic classes of Z_n, where the q-*cyclotomic class* of $j \in Z_n$ is {j,jq,jq^2,…} (the products computed mod n). The factor corresponding to the class C is the polynomial $f_C = \Pi_{j \in C}(X-\omega^j)$, where ω is a primitive n-th root of unit in a field containing K.

```
cyclotomic_class(j,n,q) :=
  begin local C, k
    if gcd(n,q)>1 then
      return show(«Number »,q,« is not prime
        to »,n) end
    j = j mod n; C = {j}
    k = (j*q) mod n
    while k != j do
      C = C | {k}
      k = (k*q) mod n
    end
    C
  end
# default: q=2
cyclotomic_class(j,n):=cyclotomic_class(j,n,2)
```

Writing a function that yields the list of all the q-cyclotomic classes presents few difficulties:

```
cyclotomic_classes(n,q):=
  begin local J, C, c;
    if gcd(n,q)>1 then
      return show(«Number »,q,« is not prime
        to »,n) end
    J={0}; C={0}
    for j in 1..(n-1) do
      if index(J,j)==0 then
        c=cyclotomic_class(j,n,q)
        C=(C,c); J=J|c
      end
    end
    {C}
  end
cyclotomic_classes(n):=  cyclotomic_classes(n,2)

cyclotomic_classes(15)
--->
{{0}, {1,2,4,8}, {3,6,12,9}, {5,10},
 {7,14,13,11}} :: List

cyclotomic_classes(15,11)
--->
{{0}, {1,11}, {2,7}, {3}, {4,14}, {5,10}, {6},
 {8,13}, {9}, {12}}
```

Since the class of 0 is {0}, and the corresponding factor is X-1, we see that X^{15}-1 factors over Z_2 into the product of five irreducible factors, one of degree 1, one of degree 2 and three of degree 4, while the same polynomial factors, over Z_{11}, into 5 linear factors and 5 quadratic factors.

One way to construct the primitive n-th root ω with OMEGA is: take r=ord(q,n); find f=polirred(K,r,T); construct F=ext(K,t,f); take a=prim(F); set $\omega = a^{(q^r-1)/n}$. We can turn this into an Ω-function:

```
omega(K:Field, n:Integer):=
  begin local q, r, f, t, F, a
    q = card(K)
    r = ord(q,n)
    if r>1 then f = polirred(K,r,t)
      F = ext(K,f)
    else F=K end
    a = prim(F)
    a^((q^r-1)/n)
  end
```

(To obtain a computation environment in which such a function definition could be evaluated was in fact one of the main motivations to start the OMEGA Project.)

The Paley construction

For finite fields K of characteristic not 2, the map $X : K^* \to \pm 1$ defined by X (a) = $a^{(q-1)/2}$ is a character (the *Legendre character*) and X (a) = 1 if and only if a is a square in K^*. We can immediately produce an Ω-function that gives this character:

```
legendre(a):= if a==0 then 0
```

```
    elif a^((card(field(a))-1)/2)==
      1 then 1
    else -1 end
```

One interesting construction that uses X is the so-called *Paley matrix* of a finite field K (see next section for an application related to codes). If the elements of K are $x_0,\ldots,x_n$, where n = q-1, then the matrix is X $(x_i\text{-}x_j)$. We can instruct OMEGA to obtain this expression as follows:

```
paley_matrix(K:Field):=
  begin
    local q, x, chi
    n=card(K)-1
    x(j):= elem(j,K)
    chi(a):=legendre(a)
    [[chi(x(i)-x(j)) with j in 0..n] with i in
      0..n]
  end

paley_matrix(Zn 5)
--->
[ 0, 1,-1,-1, 1]
[ 1, 0, 1,-1,-1]
[-1, 1, 0, 1,-1]
[-1,-1, 1, 0, 1]
[ 1,-1,-1, 1, 0]
:: Matrix(Z)
```

Note the use of the local functions x and chi, which allows us to write the final expression of the matrix in a form that is quite close to the mathematical definition.

Hadamard matrices

If we set H = H_1 to denote the matrix $\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$, and $H_n = H^{\otimes n}$, then the function hadamard below calculates the matrix H_n for any n. It is a nice example of a recursive definition, a feature that is fully supported by OMEGA. Remark that $H_1\otimes H = \begin{pmatrix}H & H\\ H & -H\end{pmatrix}$ and note the compact expression for this matrix in the function body.

```
hadamard(1):=[[1,1],[1,-1]]

hadamard(n:Integer) check n>0 :=
  begin local H
    H=hadamard(n-1)
    (H | H)
    &
    (H |-H)
  end
```

The matrices H = H_n are *Hadamard matrices*, because they only contain 1's and -1's and satisfy that $HH^T = nI_n$. Moreover, they are normalized, since the first row and column only contain 1's. These matrices are closely related to useful codes (see [20]).

```
H=hadamard(3)
--->
[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]
[1,  1, -1, -1, -1, -1,  1,  1]
[1, -1, -1,  1, -1,  1,  1, -1]
:: Matrix(Z)
```

(The evaluation of h*h' yields, as expected, $8I_8$.)

There is another construction of a (normalized) Hadamard matrix of order q+1, for any finite field K of q elements, provided that q+1 is divisible by 4 (see [20]). It uses the Paley construction and it can be programmed as follows:

```
hadamard(K:Field):=
  begin local q,u,S
    q=card(K)
    u=constantvector(1,q)

    S=idmatrix(q)+paley_matrix(K)

    ( [1] | u )
    &
    (  u  |-S )
  end

hadamard(Zn 3)
--->
[1, 1, 1, 1]
[1,-1, 1,-1]
[1,-1,-1, 1]
[1, 1,-1,-1]
```

It is to be remarked that if S is a matrix of type mxn and u a vector of dimension m, then u|S yields the matrix of type mx(n+1) obtained by attaching the column vector u' to the left of S. Likewise, if u is a vector of dimension n, then S|u yields the matrix of type mx(n+1) obtained by attaching u' to the right of S.

Cyclic codes

If $g = g_0+g_1X+\ldots+g_rX^r$ is a divisor of X^{n-1} over a finite field $\mathbb{F}_q$, and r = deg(g), then the cyclic code C_g corresponding to g (see [20]) has dimension k = n-r and the i-th row of a kxn generating matrix G has the form

$$[0,\ldots,0,g_0,\ldots,g_r,0,\ldots,0],$$

with i-1 leading zeros. Similarly, if $h = (X^{n-1})/g$, and $h = h_kX^k+\ldots+h_1X+h_0$, then the i-th row of a control matrix H that is dual to H has the form

$$[0,\ldots,0,h_k,\ldots,h_0,0,\ldots,0],$$

again with i-1 leading zeros.

The next three functions allow us to construct these matrices.

```
cyclic(g:VEC,n):=
  begin local r, k, G
    r = dim(g)-1; k = n-r
    if k>1 then g=g|constantvector(0,k-1) end
    G=g
    for i in 1..(k-1) do
```

```
      g = [0] | take(g,n-1)
      G=(G,g)
    end
    [G]
  end

cyclic_G(g:POL,n):= cyclic(pol2vector(g),n)

cyclic_H(h:POL,n):=
   cyclic(reverse(pol2vector (h)),n)
```

These functions do not check whether g is a divisor of X^n-1, nor care about the base field F_q. The point is that we will first obtain g, for example by calling `factor`(X^n-1,K) (which is, incidentally, an efficient internal function), and so the field K and the fact that g divides X^n-1 are already built-in.

The functions above can be used to construct the resultant matrix of two vectors of two univariate polynomials:

```
resultant_matrix(f:VEC,g:VEC):=
  cyclic(f,dim(f)+dim(g)-2)
  &
  cyclic(g,dim(f)+dim(g)-2)

resultant_matrix(f:POL,g:POL):=
  resultant_matrix(pol2vector(f),pol2vector(g))
```

If we want to use indeterminate coefficients to construct the corresponding resultant matrix, we need an extension `pol2vector(P,T)` of the function `pol2vector(P)` that delivers the coefficients of P as a polynomial in T:

```
pol2vector(P,T):=
  begin local c, C, k
    c = subst(P,T,0)
    C = [c]
    P = diff(P,T)
    k = 1
    while P != 0 do
      c = subst(P,T,0)
      if c!=0 then c = c/k! end
      C = C | [c]
      P = diff(P,T)
      k=k+1
    end
    C
  end
```

Now the question can be solved as follows:

```
resultant_matrix(f, g, T):=
  resultant_matrix(pol2vector(f,T),pol2vector
    (g,T))
```

Note also that the discriminant matrix of a polynomial f, with respect to T, is now given by

```
discriminant_matrix(f,T):=
  resultant_matrix(f,diff(f,T),T)
```

Let us illustrate how the above functions work in a short session (cf. [3], p. 233-4):

```
f2=T^2+b*T+c
---> T^2+Tb+c :: Z[T,b,c]

D2=discriminant_matrix(f2,T)
--->
[c, b, 1]
[b, 2, 0]
[0, b, 2]
:: Matrix(Z[b, c])

d2=det(D2)
---> -b^2+4c :: Z[b,c]

f3=T^3+a*T+b
---> T^3+Ta+b :: Z[T,a,b]

D3=discriminant_matrix(f3,T)
--->
[b, a, 0, 1, 0]
[0, b, a, 0, 1]
[a, 0, 3, 0, 0]
[0, a, 0, 3, 0]
[0, 0, a, 0, 3]
 :: Matrix(Z[a, b])

d3=det(D3)
---> 4a^3+27b^2 :: Z[a,b]


f4=T^4+a*T^2+b*T+c
---> T^4+T^2a+Tb+c :: Z[T,a,b,c]

D4=discriminant_matrix(f4,T)
--->
[c, b, a, 0, 1, 0, 0]
[0, c, b, a, 0, 1, 0]
[0, 0, c, b, a, 0, 1]
[b, 2a, 0, 4, 0, 0, 0]
[0, b, 2a, 0, 4, 0, 0]
[0, 0, b, 2a, 0, 4, 0]
[0, 0, 0, b, 2a, 0, 4]
 :: Matrix(Z[a, b, c])

d4=det(D4)
---> 16a^4c-4a^3b^2-128a^2c^2+144ab^2c-
     27b^4+256c^3 :: Z[a,b,c]
```

The Meggitt decoder

Let $g \in K[x]$ be the generating polynomial of a cyclic code C of length n over K. We want to implement the Meggitt decoder (see [13], Ch. XVII). In this decoder, a received vector $y = [y_0,\ldots,y_{n-1}]$ (note the indexing) is seen as a polynomial $y_0+y_1x+\ldots+y_{n-1}x^{n-1}$ and the *syndrome* of y is, by definition, the remainder of the euclidean division of y by g, `rem(y,g)` in OMEGA. The vectors with zero syndrome are, again by definition, the vectors of C.

If we want to correct t errors, where t is not greater than the error-correcting capacity, then the Meggitt decoding scheme presupposes the computation of a table E with the syndromes of the error-patterns of the form $ax^{n-1}+e$, where $a \in K^*$ and $e \in K[x]$ has degree n-2 (or less) and at most t-1 non-vanishing coefficients.

For example, the binary Golay code can be defined as the cyclic code of length n = 23 generated by

$$g = x^{11}+x^9+x^7+x^6+x^5+x+1 \in Z_2[x]$$

and, since in this case the error-correcting capacity is 3, the Meggitt table can be defined as follows:

```
E = [rem(x^(n-1),g)->x^(n-1)]
    +
    [rem(x^(n-1)+x^i,g)->x^(n-1)+x^i with i in
     0..(n-2)]
  +
  [rem(x^(n-1)+x^i+x^j,g)->x^(n-1)+x^i+x^j
    with (i,j) in 0..(n-2),0..(n-2) where
      j<i]
```

Actually the type of this sort of table (with brackets instead of braces) is `Divisor`. Divisors can be added, and the result is like joining the tables, but adding the values for the keys that appear in both addends. The advantage of divisors D over tables for our present problem is that D(s) returns 0 if s is not in the domain of D. Thus, for the divisor E above, E(s) is 0 for all the syndromes s that do not coincide with the syndrome of x^{22}, or that of $x^{22}+x^i$ for i = 0, …, 21, or that of $x^{22}+x^i+x^j$ for i,j ∈ {0,1,…,21} and i > j. Otherwise E(s) selects, among those polynomials, the one that has syndrome s.

Now we implement the Meggitt decoder as an Ω-function `meggitt(y,g,n)` with parameters y, the polynomial to be decoded, g, the polynomial generating the code, and n, the length. The algorithm is as follows:

- Find the syndrome $s = s_0$ of y.
- If s vanishes, y is a code vector and we return y.
- Otherwise compute, for j = 1,2,…,n-1, the syndromes s_j of $x^j y$, and stop for the first j such that e = E(s) ↑ 0.
- At this point we know that the symbol of degree n-1-j in y is an error and that the error is the leading coefficient of e. Thus we can correct this error, if c is the leading coefficient of e, by replacing y by $y-cx^{n-1-j}$.
- Apply the same procedure to the new y.

```
meggitt(y,g,n):=
  begin local x, s, j
    x = var(g)         # the variable of g
    s = rem(y,g)       # the syndrome
    if s == 0 then return
      show(«Code word: \n»,y) end
    j = 0
    while E(s) == 0 do
      j = j+1
      s = rem(x^j*y,g)
    end
    y = y - lcoef(E(s))*x^(n-1-j)
    show(«Error in degree », n-1-j, « is
      corrected»)
    meggitt(y,g,n)
  end
```

The following example shows how this decoder works for the binary Golay code:

```
Z2= Zn 2;  u=1:Z2
n=23

clear x
g=x^11+x^9+x^7+x^6+x^5+x+u  # generator of
                            # Golay2

# Construction of Meggitt table E. We do not
# list it again: see the first listing
# in the last section

y = x^20+x^15+x^10+x^5+u

meggitt(y,g,n)
--->
Error in degree 14 is corrected
Error in degree 12 is corrected
Error in degree 4 is corrected
Code word:
x^20+x^15+x^14+x^12+x^10+x^5+x^4+1 :: Z2[x]
```

The ternary Golay code can be defined as the cyclic code of length 11 generated by

$$g = x^5+x^4+2x^3+x^2+2 \in Z_3[x]$$

and in this case, since the error-correcting capacity is 2, the Meggitt table can be defined as follows (with n = 11 and z3 = Z_3-{0}):

```
E =
[rem(a*x^(n-1),g)->x^(n-1) with a in z3]
+
[rem(a*x^(n-1)+b*x^i,g)->a*x^(n-1)+b*x^i
      with (i,a,b) in (0..(n-2),z3,z3)]
```

Now we can work out examples like the following:

```
Z3= Zn 3; u=1:Z3
n=11
z3=[u,2]

clear x
g=x^5+x^4+2*x^3+x^2+2*u

# Construction of Meggitt table, as explained
# above. We do not repeat it here.

y = x^4+x^3+x+u

meggitt(y,g,n)
--->
Error in degree 9 is corrected
Error in degree 5 is corrected
Code word:
x^9+x^5+x^4+x^3+x+1 :: Z3[x]
```

The factors to construct the Golay codes can be obtained by the internal function `factor(f,K)`:

```
factor(x^23-1,Zn 2)
--->
```

```
{{x+1,1},{x^5+x^4+2x^3+x^2+2,1}},
 {x^5+2x^3+x^2+2x+2,1}}

factor(x^11-1,Zn 3)
--->
{{x+2,1}, {x^11+x^9+x^7+x^6+x^5+x+1,1}},
 {x^11+x^10+x^6+x^5+x^4+x^2+1,1}}
```

Since the Golay codes are perfect, it follows that the Meggitt decoder is complete for these codes.

MacWilliams identities

A nice example of polynomial manipulation is calculating the weight enumerator of the dual of a linear code given the weight enumerator of the code (cf. [17], p. 225). If the weight enumerator of our code is A, n its length and k its dimension, then the function `macwilliams` returns the weight enumerator of the dual code:

```
macwilliams(A,n,k,q):=
  (1+(q-1)*t)^n * subst(A,t,((1-t)/(1+
    (q-1)*t)))/q^k
macwilliams(A,n,k):=
  (1+t)^n*subst(A,t,(1-t)/(1+t))/2^k

macwilliams(1+t^3,3,1)
---> 1+3t^3

macwilliams(1+(2^3-1)*t^4,7,3)
---> 1+7t^3+7t^4+t^7
```

The latter, for example, is the weight enumerator of the Hamming [7,4,3] binary code, obtained from the weight enumerator $1+7t^4$ of its dual code.

Cyclotomic polynomials

Let us present here the computation of the cyclotomic polynomials with OMEGA. The algorithm used to obtain `Phi(n,T)`, the n-th cyclotomic polynomial over $\mathbb{Z}$, is due to H. Lüneburg and is taken from [7] (Algorithm 2.6.7, p. 73; see also references there).

```
Phi(n,T) :=
  begin local P, # to hold the canonical
                 # factorization of n
       s, # the number of prime factors of n
       i, # current prime factor index
       p, # current prime factor
       m, # current product of prime divisors
          # of n
       f # current value of result
    P = factor(n); s=dim(P)
    p=P.1.1; m=p
    f = sigma T^(p-i) with i in 1..p # Phi(p,T)
    i=1
    while i<s do
      i=i+1; p=P.i.1; m=m*p;
      f = eval(f,T^p)/f
    end
    eval(f,T^(n/m))
  end
```

For example, the expression

```
Phi(4725,T)
--->
T^2160+T^2115+T^2070-T^1935-T^1890-2T^1845-
T^1800-T^1755+T^1620+T^1575+T^1530+T^1485+
T^1440+T^1395-T^1260-T^1170-T^1080-T^990-T^900+
T^765+T^720+T^675+T^630+T^585+T^540-T^405-
T^360-2T^315-T^270-T^225+T^90+T^45+1 :: Z[T]
```

is evaluated in 0.06 s, and the expression

```
{Phi(n,T) with n in 1..100 where
 not(prime?(n))}
```

in 0.44 s (75 polynomials).

Primitive irreducible polynomials

Given a monic irreducible polynomial $f \in K[T]$ of degree r, K being a finite field, the order of the class t = [T] of T mod f is often a proper divisor of q^{r-1} ($q = |K|$). In other words, t need not be a primitive element of F = K[T]/(f). The order of t is also said to be the *order of* f, and f is said to be *primitive* if and only if t is a primitive element of F. The order of f can thus be calculated by the following function:

```
ord(p,K):=
  begin local x, f, t, F
    x=var(p)
    f=subst(p,x,t)
    F=ext(K,f)
    ord(t)
  end
```

We usually need to choose f primitive to start with. Therefore we need a function that returns, given a field K, a degree r and an indeterminate T, a monic irreducible primitive polynomial of degree r over K.

```
primitive_irr_pol(K,r,T):=
  begin local f, t, F, x
    f=polirred(K,r,t)
    F=ext(K,f)
    x=prim(F)
    minpol(x,K,T)
  end
```

Although we rarely will need it, we could as well want the list of all the monic irreducible primitive polynomials of a given degree r over K. This can be done by the internal function `listirred(K,r,T)`, which gives the list of all the monic irreducible polynomials of degree r over K, and the function `ord(p,K)` explained above:

```
all_primitive_irr_pols(K,r,T):=
  {p where ord(p,K)==card(K)^r-1 with p in
   listirred(K,r,T)}
```

These functions are reasonably efficient. For example, to find one irreducible primitive polynomial of degree i over $\mathbb{Z}_2$, for each i in the range 2..30, it took 18 s, and 341 s to find the 1800 primitive polynomials of degree 15 over $\mathbb{Z}_2$ (the

number of irreducible polynomials of degree 15 over Z_2 is 2182).

The other approach, namely factoring $\Phi(q^{r-1},T) \in K[T]$ (its factors are precisely the monic irreducible primitive polynomials of degree r over K) becomes cumbersome because the degree of the cyclotomic polynomial becomes immediately very large when we increase r.

Krawtchouk polynomials

Mathematically, these polynomials are defined by the expression

$$\mathrm{Kr}(X,k;n,q) = \sum_{j=0}^{k} (-1)^j \binom{X}{j} \binom{n-X}{k-j} (q-1)^{k-j}$$

(see [20], §1.2). We see that this expression is a polynomial of degree k in X whose coefficients are themselves polynomials in n and q. For their significance in the theory of error correcting codes, we refer to [20], especially §5.3.

With OMEGA we can get them as indicated below.

```
newton(x,j):=
   (product (x-i+1) with i in 1..j)/j!

Kr(x,k,n,q):=sigma (-1)^j*newton(x,j)*newton
  (n-x,k-j)*(q-1)^(k-j) with j in 0..k
Kr(x,k,n):= sigma (-1)^j*newton(x,j)*newton
  (n-x,k-j) with j in 0..k

Kr(X,0,n,q)
---> 1 :: Z
Kr(X,1,n,q)
---> -Xq+nq-n :: Z[X,n,q]
Kr(X,2,n,q)
--->
1/2(X^2q^2 + X(-2nq+2n+q-2)q + n^2q^2-
  2n^2q+n^2-nq^2+2nq-n)
:: Q[X,n,q]
Kr(X,3,n,q)
--->
-1/6X^3q^3 + 1/2X^2(nq-n-q+2)q^2 - ...
:: Q[X,n,q]
```

Let us check the orthogonality relation

$$\sum_{i=0}^{n} \mathrm{Kr}_l(i)\mathrm{Kr}_l(k) = \delta_{lk} q^n$$

(see [20], equation 1.2.6) for n = 10 and q = 2:

```
n=10
V=[[subst(Kr(x,l,n),x,i) with i in 0..n] with
   l in 0..n]
--->
[  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,   1]
[ 10,  8,  6,  4,  2,  0, -2, -4, -6, -8, -10]
[ 45, 27, 13,  3, -3, -5, -3,  3, 13, 27,  45]
[120, 48,  8, -8, -8,  0,  8,  8, -8,-48,-120]
[210, 42,-14,-14,  2, 10,  2,-14,-14, 42, 210]
[252,  0,-28,  0, 12,  0,-12,  0, 28,  0,-252]
[210,-42,-14, 14,  2,-10,  2, 14,-14,-42, 210]
[120,-48,  8,  8, -8,  0,  8, -8, -8, 48,-120]
[ 45,-27, 13, -3, -3,  5, -3, -3, 13,-27,  45]
[ 10, -8,  6, -4,  2,  0, -2,  4, -6,  8, -10]
[  1, -1,  1, -1,  1, -1,  1, -1,  1, -1,   1]
:: Matrix(Z)

W=[[subst(Kr(x,i,n),x,k) with i in 0..n] with
   k in 0..n]
--->
[1,  10, 45,  120, 210,  252, 210, 120,  45,  10,  1]
[1,   8, 27,   48,  42,    0, -42, -48, -27,  -8, -1]
[1,   6, 13,    8, -14,  -28, -14,   8,  13,   6,  1]
[1,   4,  3,   -8, -14,    0,  14,   8,  -3,  -4, -1]
[1,   2, -3,   -8,   2,   12,   2,  -8,  -3,   2,  1]
[1,   0, -5,    0,  10,    0, -10,   0,   5,   0, -1]
[1,  -2, -3,    8,   2,  -12,   2,   8,  -3,  -2,  1]
[1,  -4,  3,    8, -14,    0,  14,  -8,  -3,   4, -1]
[1,  -6, 13,   -8, -14,   28, -14,  -8,  13,  -6,  1]
[1,  -8, 27,  -48,  42,    0, -42,  48, -27,   8, -1]
[1, -10, 45, -120, 210, -252, 210,-120,  45, -10,  1]
:: Matrix(Z)

V*W'
--->
[1024, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1024, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1024, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1024, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1024, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1024, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1024, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1024, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 1024, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1024, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1024]
:: Matrix(Z)
```

Rational points on plane curves

Let us give some examples of how to obtain the rational points of an affine plane curve over a finite extension of the base field.

The first example is about the F_8-rational points of the Klein quartic $x^3y+y^3+x = 0$ over Z_2.

```
Z2=Zn 2

f(x,y):=x^3*y+y^3+x

clear x
F8=ext(Z2,x^3+x+1)
n=card(F8)-1
f8 = [0] | [x^i with i in 0..n-1]

X = [[x,y] where f(x,y)==0
        with (x,y) en (f8,f8)]
--->
[[0, 0], [1, x], [1, x^2], [1, (x^2+x)],
 [x, 1], [x, x^2], [x, (x^2+1)], [x^2, 1],
 [x^2, (x^2+x)], [x^2, (x^2+x+1)],
```

```
 [(x+1), x^2], [(x+1), (x+1)],
 [(x+1), (x^2+x+1)], [(x^2+x), 1],
 [(x^2+x), x], [(x^2+x), (x+1)],
 [(x^2+x+1), x], [(x^2+x+1),(x2+x+1)],
 [(x^2+x+1), (x^2+1)], [(x^2+1), (x+1)],
 [(x^2+1), (x^2+x)], [(x^2+1), (x^2+1)]]
:: Matrix(F(2^3))
```

The second example shows how to find the F_{64}-rational points of the affine plane curve $x^9+y^8+y = 0$, defined over Z_2 (for this and other examples, see [18]). The procedure is very similar to the one followed in the previous example. However, since there are 512 points (it takes about 7 s to calculate them), we will not reproduce them here.

```
g(x,y):= x^9+y^8+y

Z2 = Zn 2

clear t
F64=ext(Z2,t^6+t+1)

q=card(F64)

f64=[0] | [t^i with i in 0..(q-2)]

Y=[[x,y] where g(x,y)==0
        with (x,y) in (f64,f64)]

dim(Y)
---> 512 :: Z
Y.100
---> [t^5+t+1,t^5+t^3+1] :: Vector(F(2^6))
```

Rational functions

Among the fundamental services needed for computational work in the field of algebro-geometric Goppa codes we have to mention the capacity to define rational functions on any number of variables and a procedure for evaluating them at any point. OMEGA implements these services and so, for example, the rational function $f = (u^2v+uv^2+2)/(u^{15}+uv^7-7)$ can be evaluated at the point [2,3] as follows:

```
clear u, v
f = (u^2*v+u*v^3+2)/(u^15+u*v^7-7)
P = [2,3]
eval(f,P)
---> 68/37135 :: Q
```

Thus, we have the tools to construct the matrix of values of a list or vector of rational functions on a list or vector of rational points, an object that is of essential in the field. If f is the vector of rational functions and P is the vector of points, the matrix can be obtained with the Ω-expression

```
[[eval(f.i,P.j) with j in range(P)] with i in
 range(f)]
```

Let us illustrate this with an example (cf. [15], example of §5.5). The listing does not contain the output for H (a matrix of type 18x14 over F_{16}), but otherwise is quite explicit.

```
Z2=Zn 2  # base field

clear x
F=ext(Z2,x^4+x^3+1)  # F16

f16=[0]|[x^i with i in 0..(card(F)-2)]

f(x,y):=x^3*y+y^3+x  # Klein quartic

# F16-rational points of the curve
P={{a,b} where f(a,b)==0
        with (a,b) in (f16,f16)}
P=tail(P)# we discard the origin, a pole for the
         # functions f below
dim(P)
---> 14 :: Z

clear X, Y

# List of 18 rational functions
f = {1,1/X,Y/X^2,1/X^2,Y^2/X^3,Y/X^3,1/X^3,
     Y^2/X^4,Y/X^4,1/X^4,Y^2/X^5,Y/X^5,1/X^5,
     Y^2/X^6,Y/X^6,1/X^6,Y^2/X^7,Y/X^7}

H=[[eval(f.i,P.j) with j in range(P)]
        with i en range(f)]

H.7.11
---> x+1 :: F(2^4)
eval(f.7,P.11)
---> x+1 :: F(2^4)

rang(H)
---> 14 :: Z
```

Two conerstone functions for intersection theory

First we introduce two auxiliary functions. If $c = [c_1,\ldots,c_n]$ is the vector of Chern classes of some bundle, the dual bundle has Chern classes $(-1)^i c_i$. Let us define an Ω-function for producing these dual classes:

```
dual(c):= [(-1)^i*c.i with i in range(c)]
```

The other auxiliary function is `convolution(u,v,k)`, where u and v are vectors and k is an integer. It will be clear that it returns the sum $u_1v_{k-1}+\ldots+u_{k-1}v_1$, with the convention that u_i (v_j) is taken to be zero if i > dim(u) (j > dim(v)), or if $i \leqslant 0$ ($j \leqslant 0$).

```
convolution(u,v,k):=
  sigma u.i * v.(k-i)
    where i<=dim(u) & (k-i)<=dim(v)
    with i in 1..(k-1)
```

The two functions to which this section is devoted are c2p and p2c, which transform, respectively, the total Chern class $c = [c_1,c_2,\ldots,c_n]$ into the total Chern character $p = [p_1,p_2/2!,\ldots,p_n/n!]$ and conversely. We just apply the formulae on p. 56 of [5]. In any case, these functions are cornerstones of the Ω-library for intersection theory computations (see [25]), like the Maple functions expp and logg in the Schubert package of S. Katz and S. A. Strømme.

```
c2p(c,r:Integer):=
begin
  local p, sk
```

```
  p=[c.1]
  c=dual(c)
  for k in 2..r do
    sk = convolution(c,p,k)
    sk= -( k*c.k + sk )
    p = p | [sk]
  end
  [p.k/k! with i in range(p)]
end

p2c(p,r):=
begin
  local c, sk
  p=[k!*p.k with k in range(p)]
  c=[-p.1]
  for k in 2..r do
    sk = convolution(c,p,k)
    sk= -( p.k + sk )/k
    c = c | [sk]
  end
  dual(c)
end

c=[c1,c2,c3,c4]
c2p(c,4)
--->
[c1, 1/2(c1^2-2c2), 1/6(c1^3-3c1c2+3c3),
 1/24(c1^4-4c1^2c2+4c1c3+2c2^2-4c4]
 :: Vector(Q[c1, c2, c3, c4, c5, c6, c7, c8])

p=[x1,x2,x3,x4]
p2c(p,4)
--->
[x1, 1/2(x1^2-2x2), 1/6(x1^3-6x1x2+12x3),
 1/24(x1^4-12x1^2x2+48x1x3+12x2^2-144x4]
 :: Vector(Q[x1, x2, x3, x4, x5, x6, x7, x8])
```

Since the relation between the total Chern class $c = [c_1,c_2,\ldots,c_n]$ and the Chern character $p = [p_1,p_2,\ldots,p_n]$ is that $k!p_k$ is the k-th Newton sum of the roots of the polynomial $x^n - c_1x^{n-1}+\ldots+(-1)^nc_n$, one application of the functions above is the calculation of the Newton sums of the roots of a given polynomial in terms of its coefficients:

```
newton_sums(f,T,r):=
  begin local c, n, p
    c=pol2vector(f,T)
    n=dim(c)-1
    c=take(c,n)     # discard the leading coef 1
    c=dual(reverse(c))
    if r>dim(c) then
       c=c|constantvector (0,r-dim(c)) end
    p=c2p(c,r)
    [k!*p.k with k in range(p)]
  end

f=x^4+a*x^2+b*x+c

newton_sums(f,x,10)
--->
[0, -2a, -3b, 2a^2-4c, 5ab, -2a^3+6ac+3b^2,
 -7a2b+7bc, 2a^4-8a^2c-8ab^2+4c^2,
 9a^3b-18abc-3b^3,
 -2a^5+10a^3c+15a^2b^2-10ac^2-10b^2c]
 :: Vector(Z[a, b, c])
```

For an interesting application of these sums to the problem of finding the number of real roots of a real polynomial, see [16].

7. Ending remarks

We have seen that OMEGA can evaluate very abstract logic and mathematical expressions, which are syntactically close to familiar formulas, like

```
P = {{a,b} suchthat f(a,b)==0 with (a,b) in F^2}
```

On the other hand, all the objects in Ω (including types and functions) are first class, in the sense that any object can be passed as a parameter to a function, or returned by other functions, or assigned to a local variable. Thus Ω has high mathematical and algorithmic expressive power (as shown in the diverse examples presented in these notes), while at the same time it provides an efficient way to run the algorithms and to organize them in libraries.

Such features make OMEGA a powerful tool for analyzing and solving problems. In these notes, this has been illustrated for the case of error-correcting codes (see also [26]), but it happens likewise in other areas, as is explained in more detail in some works in progress (cf. [25]). See also [27].

Our next aims are to improve the user interface, such as high quality mathematical printing, both for the output and for the input of expressions; to reinforce its analytical and numerical modules; and to improve the geometric and graphic services.

One final point is about the multilingual capacity of OMEGA. There are two main aspects of this facility. One is related to commands of the form

```
babel «latin»
```

Such commands can be inserted in Ω-files, but presuppose that a file `latin.dic` is available. Its effect is that we can use, until another `babel` command or the end of file is found, the reserved words whose translation into the default reserved words is included in `latin.dic`. The default reserved words are in English, and can always be used anywhere. For example, with a dictionary like

```
# Latin.dic file
cum -> with
talisut -> suchthat
primus? -> prime?
```

we could write a file like

```
# Programming in latin
babel «latin»
n=10000
p={k talisut primus?(k) cum k in 1..n}
```

which would be evaluated as

```
# Programming in English
n=10000
p={k suchthat prime?(k) with k in 1..n}
```

Note that in is the same in Latin as in English, and so cannot be included in the latin dictionary. A dictionary can contain multiple translations of the same default reserved word and users can write their own dictionaries.

The other aspect is that the user can select in the options menu a language L, which activates the command `babel` «L» at the command line, so that the user can type expressions in language L. Another effect of this selection is that the output on the screen will be in the language L, both for the values of the expressions entered from the keyboard or entered by loading a file.

Acknowledgements

OMEGA is also the name of a group of technological projects developed, or under development, at the FME of the UPC, under the direction of the author (see [22], [10], [1], [4], [2], [6]).

For the programming of the system, the OMEGA team has used facilities of MA2, especially its computer lab.

The UPC has also assisted through the "Programa Innova'', which has had a very positive effect on the OMEGA team. Through this programme it has been possible to learn many important capacities which were far from the academic makeup of the team members and to give a deeper sense of purpose to its efforts.

I also thank Leonor for her tact, care and patience, especially during the preparation of this work.

This work has been partially supported by the DGICYT research grants PB94-1196 and TIC99-0762 002-01.

References

[1] D. Arso, Projecte OMEGA: *Cossos finits i polinomis*. FME/UPC, September 1998.

[2] M. Castells, Projecte OMEGA: *El mòdul geomètric i gràfic*. In preparation.

[3] F. Delgado, C. Fuertes, S. Xambó, *Introducción al álgebra*, vol. 2. Univ. de Valladolid, 1998.

[4] R. Eixarch, Projecte OMEGA: el mòdul d'enters i racionals. FME/UPC, July 1999.

[5] W. Fulton, *Intersection theory* (2nd ed.). Ergebnisse 2 (3. Folge), Springer-Verlag, 1998.

[6] P. Garriga, Projecte OMEGA: *El mòdul d'àlgebra lineal*. In preparation.

[7] D. Jungnickel, *Finite Fields -Structure and Arithmetics*. B.I. Wissenshaftsverlag, 1993.

[8] R. Lidl and H. Niederreiter, *Introduction to Finite Fields and their Applications* (revised edition). Cambridge University Press, 1994.

[9] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-correcting Codes*. North-Holland Mathematical Library 6, North-Holland, 1977.

[10] D. Marquès, Projecte OMEGA: MEGA *-un prototipus de manipulador simbòlic*. FME/UPC, June 1998.

[11] R. J. McEliece, *Finite Fields for Computer Scientists and Engineers*. Kluwer Academic Publishers, 1989 (second printing).

[12] A. J. Menezes (editor), *Applications of finite fields*. Kluwer Academic Publishers, 1993.

[13] O. Papini, J. Wolfmann, *Algèbre discrète et codes correcteurs*. Springer-Verlag, 1995.

[14] A. Poli, Ll. Huguet, *Codes correcteurs -théorie et applications*. Masson, 1989.

[15] O. Pretzel, *Codes and Algebraic Curves*. Oxford Lect. Ser. in Math. 8, Oxford Science Publications, 1998

[16] T. Recio, *La columna de matemática computacional*, La Gaceta de la RSME, vol. 1, no. 1, 105-111.

[17] S. Roman, *Coding and Information Theory*. GTM 134, Springer-Verlag, 1992.

[18] M. A. Shokrollahi, H. Wasserman, *List Decoding of Algebraic-Geometric Codes*. IEEE Trans. Inf. Theory, 45, no. 2, March 1999, 432-437.

[19] M. A. Tsfasman and S. G. Vl\breveadut, *Algebraic-Geometric Codes*. Mathematics and Its Applications (Soviet series), Kluwer Academic Publishers, 1991.

[20] J. H. van Lint, *Introduction to Coding Theory* (3rd edition). GTM 82, Springer-Verlag, 1999.

[21] J. H. van Lint and G. van der Geer, *Introduction to Coding Theory and Algebraic Geometry*. DMV Seminar 12, Birhäuser Verlag, 1988.

[22] X. Vindel, Projecte OMEGA: *Teoria computacional de grups*. FME/UPC, May 1998.

[23] S. B. Wicker, V. K. Bhargava (eds.), *Reed-Solomon codes and their applications*. The Institute of Electrical and Electronics Engineers, 1994.

[24] S. Xambó, *Using Intersection Theory*. Aportaciones Matemáticas (7, nivel avanzado), Sociedad Matemática Mexicana, 1996.

[25] S. Xambó, WIT: *An OMEGA Library for Intersection Theory*. In preparation.

[26] S. Xambó, M. Bras, *OMEGA decoders of Goppa codes*. In preparation.

[27] S. Xambó *et al.*, OMEGA/Athens: *Users Manual*/1999.

[28] S. Xambó, D. Marquès, *Mathematical symbolic systems and functional languages*. Proceedings of the "IV Journées Catalanes de Mathématiques Appliquées», 11-13 February 1998, Tarragona (C. Garcia, C. Olivé, M. Sanromà eds.), 175-192.

[29] S. Xambó and J. Mola, OMEGA: *Engineering a System for Effective Computations related to Block Error-correcting Codes*. Proc. III International Congress of Project Engineering, eds. A. Creus, J. Forès, F. Tadeusz and J. Aragonès (AEIPRO, 1996), 1518-1525.

[30] S. Xambó and J. Mola, OMEGA: *Problem Solving with the Interpreter* ω_0. Proc. III International Congress of Project Engineering, eds. A. Creus, J. Forès, F. Tadeusz and J. Aragonès (AEIPRO, 1996), 1697-1704.