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 $\mathbb{F} = \mathbb{F}_q$.
- Polynomials and matrices over $\mathbb{F}$.
- Algebraic curves $X$ over $\mathbb{F}$.
- The points on $X$ that are rational over $\mathbb{F}$.
- 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 $\Omega$) will gradually emerge. To be more specific, in 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-
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 \( \Omega \), 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 \( \Omega \)).

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

\[
\text{User}[n]: \ |
\]

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

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

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

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

Anyhow, the answer is

\[
\text{OMEGA}[127] := \{2,3,5,\ldots,9973\} :: \text{List}
\]

\[
\text{User}[128] : |
\]

(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:

\[
\text{bye();}
\]

\[
\text{exit();}
\]

\[
\text{quit();}
\]

Types

The language \( \Omega \) is typed. To indicate that a value \( x \) has type \( T \), we (and OMEGA) write \( x :: T \). Thus, the type of \( p \) above is \( \text{List} \) (by definition, a \textit{list} is a finite sequence of objects enclosed within braces). The length of a list \( L \) can be obtained writing \( \text{dim}(L) \) or \( \text{length}(L) \). 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 \textit{Integer} (written \( \mathbb{Z} \) in the output). For example, \( p.100 \) denotes the 100-th term in the list \( p \), and we have:

\[
\text{User}[128]: p.100;
\]

\[
\text{OMEGA}[128] := 541 :: \mathbb{Z}
\]

Integers in \( \Omega \) are roughly bounded to \( 4 \times 10^{10} \) decimal digits. Thus, two digits of \( 2 \times 10^{10} \) decimal digits can be multiplied. Or, theoretically, we could find \( n! \) for \( n = 5 \times 10^8 \), 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 \( \text{dim}(v) \) or \( \text{length}(v) \). 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:

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

\[
w = u|v
\]

\[
---\rightarrow [1,2,3,a,b,c] :: \text{Vector}(\mathbb{Z}[a,b,c])
\]

\[
u |-u|w|-v
\]

\[
---\rightarrow [1,2,3,-1,-2,-3,1,2,3,a,b,c,-a,-b,-c] :: \text{Vector}(\mathbb{Z}[a,b,c])
\]

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 \( A^T \) (or \( \text{transpose}(A) \)). 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 \textit{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 \textit{latent
way of referring to the integers \(a, a+c, a+2c, \ldots\) 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 \([a, a+c, a+2c, \ldots]\). Here is an example:

\[
\begin{align*}
 r &= 34 \ldots 180 \ldots 17; \\
 \Rightarrow &= 34 \ldots 180 \ldots 17 \text{ :: Range} \\
 [r] &= [34, 51, 68, 85, 102, 119, 136, 153, 170] \\
 &\text{ :: Vector(Z)}
\end{align*}
\]

(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 \(\Rightarrow\) in between).

Functions

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

\[
\text{binomial}(n, m)
\]

that yields the binomial number \(^{n\text{ \choose m}}\),

\[
\text{prim}(K)
\]

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

\[
\text{polirred}(K, r, T)
\]

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 \(\text{binomial}(n, m)\) has two arguments, which are supposed to be non-negative integers. The function \(\text{prim}(K)\) has a unique argument, which is a (finite) field. And the function \(\text{polirred}(K, r, T)\) 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,

\[
rv(n, m) := [\text{random}(m) \text{ with } i \in 1 \ldots n]
\]

is an external function that constructs a vector of length \(n\) whose components are integers in the range \(0 \ldots (m-1)\), each taken at random. Notice that here we use the internal function \(\text{random}\) which returns, given a positive integer \(m\), a pseudo-random integer in the range \(0 \ldots (m-1)\). (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 \(\text{read}\) (respectively \(\text{load}\)). We can call such a file an \(\Omega\)-library. By default, its extension is assumed to be \(\text{.cm}\) (.cmd also works if no file \(\text{.cm}\) can be found, but now this extension is considered obsolete). Thus,

\[
\text{read «tools»;}
\]

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

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

\[
\begin{align*}
\# \text{DEG-RAD.COMD} \\
\# \text{To transform radians into degrees} \\
\text{rad2deg}(r) &:= r \times \frac{180}{\pi}; \\
\# \text{To transform degrees into radians} \\
\text{deg2rad}(d) &:= d \times \frac{\pi}{180}; \\
\# \text{Radians } r \text{ into } [d, m, s] \leftarrow d \text{ degrees,} \\
\# \text{m minutes, s seconds} \\
\text{rad2dms}(r) &:= \begin{align*}
\text{begin local } &x, d, m, s; \\
&x = \text{rad2deg}(r); \\
&d = \text{floor}(x); \\
&\# \text{floor}(x) = \text{the integral part of } x \\
&x = (\text{decimal}(x) \times 60); \\
&\# \text{decimal}(x) = x - \text{floor}(x) \\
&m = \text{floor}(x); \\
&\# \text{decimal}(x) = x - \text{floor}(x) \\
&s = \text{floor}(x); \\
\end{align*} \\
&[d, m, s]
\end{align*}
\]

This file can be read with the command

\[
\text{read «deg-rad»}
\]

or with the command

\[
\text{load «deg-rad»}
\]

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

\[
\begin{align*}
\text{sin(30°)} &\Rightarrow 0.500
\end{align*}
\]

Functions, as any other object in OMEGA, need not be bound to a name. Thus \(x \rightarrow 1/x\) 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

\[ f := x \rightarrow 1/x \]

The value of both assignments is

\[ x \rightarrow 1/x :: \text{Function} \]

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

\[ \text{inv}(v) := \text{map}(x\rightarrow 1/x,v) \]

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

\[ \text{inv}([1..5]) \quad \longrightarrow \quad [1,1/2,1/3,1/4,1/5] :: \text{Vector(Q)} \]

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

\[ f(x) \]

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

\[ \text{inv}[1..5] \]

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

Polynomials

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

\[ \text{vector2pol}(a,T) \]

returns the polynomial \( a_1+a_2T+\cdots+a_nT^{n-1} \), whereas

\[ \text{roots2pol}(a,T) \]

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

\[
\text{product}(T-a_i) \text{ with } i \text{ in range(a)}
\]

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

\[ \text{diff}(f,x) \]

gives the derivative of \( f \) with respect to \( x \).

\[
\text{diff}(5*x^14+13*x^7*y^31,y);
\]

\[ \longrightarrow \quad 403 \times^7 y^30 \]

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 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 $(p-1)/\text{ord}(x)$ 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 $p-1$)

$y=x^{(p-1)/\text{ord}(x)} \rightarrow 1118708644 :: K$

will be an element of order 3607:

$\text{ord}(y) \rightarrow 3607 :: Z$

As already mentioned, $\text{prim}(K)$ returns a primitive element of $K$:

$\text{prim}(2p) \rightarrow 3 :: Z(1234567891)$

$z=(3:K)^{(p-1)/3607} \rightarrow 1058784230 :: K$

$\text{ord}(z) \rightarrow 3607 :: Z$

Construction of extensions

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

$\text{ext}(K,t,f)$

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 $a_0+a_1t+\ldots+a_rt^r$, where $r$ is the degree of $f$ and $a_i \in K$. The same result can be obtained by the function call

$\text{ext}(K,f(t))$

In order to make the function $\text{ext}$ practical, we need a way to find irreducible polynomials over $K$. This is accomplished by the function $\text{polirred}$. More specifically,

$\text{polirred}(K,r,T)$

yields a monic polynomial of degree $r$ with coefficients in $K$ in the indeterminate $T$ and which is irreducible over $K$. For example, if $K=\mathbb{Z}_n$, the following call produces a list of 16 monic irreducible polynomials over $K$ with successive degrees in the range 2..17:

$\{\text{polirred}(K,i,T) \text{ with } i \text{ in } 2..17\} \rightarrow \{T^2+14, T^3+T+3, T^4+14, T^5+T+3, T^6+T+7, T^7+T+5, T^8+14, T^9+T+3, T^{10}+T+7, T^{11}+3T+7, T^{12}+T+2, T^{13}+2T+6, T^{14}+T+8, T^{15}+3T+6, T^{16}+14, T^{17}+16T+16\}$

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

$F=\text{ext}(K,t^17-t-1) \rightarrow F(17^{17}) :: \text{Field}$

And now we can operate in $F$. For example:

$n=\text{card}(F)-1 \rightarrow 82742026188636764176 :: Z$

$r=\text{ord}(t) \rightarrow 5170251636789647761 :: Z$

$n/r \rightarrow 16 :: Z$

$\text{minpol}(t,k,T) \rightarrow T^{17}+16T+16 :: Z_{17}[T]$

$\text{minpol}(t^{700},k,T) \rightarrow T^{17}+7T^{15}+16T^{14}+12T^{12}+13T^{11}+7T^{10}+3T^9+16T^8+2T^7+4T^6+8T^5+12T^4+12T^3+8T^2+16 :: Z_{17}[T]$

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

$\text{listirred}(K,r,T)$

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

Two more internal functions worth being mentioned are

$\text{factor}(f,K)$

$\text{irred}(f,K)$

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

$\{a->x,b->y,\ldots\}$
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, ... behave 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 \( i < j \).

\[
\begin{align*}
\text{L} &= \{(i^3+j^3 \rightarrow (i,j) \text{ with } (i,j) \text{ in } (1..30)^2 \text{ where } i < j\} \\
\text{The result is a table of dimension 465 and so we have omitted it. Instead, we have selected the elements of the table that have more than one decomposition:} \\
\text{select(L, (x,y) \rightarrow \text{nops(L(x))}>1) \rightarrow } \\
&\begin{cases}
1729 \rightarrow \{(1, 12), (9, 10)\}, \\
4104 \rightarrow \{(2, 16), (9, 15)\}, \\
13832 \rightarrow \{(2, 24), (18, 20)\}, \\
20683 \rightarrow \{(10, 27), (19, 24)\}.
\end{cases}
\end{align*}
\]

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 \rightarrow a, 2 \rightarrow b, 1 \rightarrow c\} \rightarrow \{1 \rightarrow (a, c), 2 \rightarrow 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:

\[
\text{L} = \{(i^3+j^3 \rightarrow (i,j) \text{ with } i, j \text{ in } 1..30, 1..30 \text{ such that } i < j\}
\]

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 \((r, r^2)\), or just \( r, r^2 \).

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 \( x \), \( \text{support}(x) \), is the set of the indices \( i \in \{1, \ldots, n\} \) such that \( x_i \neq 0 \).

We set

\[
\text{weight}(x) = |x| = |\text{support}(x)|
\]

(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 \( x_i \neq y_i \), and we say that it is the Hamming distance between \( x \) and \( y \).

It is easy to write \( \Omega \)-functions, which we will call support, weight and dist, that implement the functions above.

\[
\begin{align*}
\text{support}(x) &:= \{i \text{ where } x_i \neq 0 \text{ with } i \text{ in } \text{range}(x)\}, \\
\text{weight}(x) &:= \dim(\text{support}(x)), \\
\text{dist}(x, y) &:= \text{weight}(x-y)
\end{align*}
\]

In the following examples, we use the function \( \text{rv}(n, m) \) introduced in section 1.4:

\[
\begin{align*}
x &= \text{rv}(8, 2) \rightarrow \{0, 1, 1, 0, 0, 0, 1, 0\} :: \text{Vector}(Z) \\
y &= \text{rv}(8, 2) \rightarrow \{0, 0, 1, 1, 1, 1, 1, 0\} :: \text{Vector}(Z) \\
\text{weight } x &\rightarrow 3 :: Z \\
\text{weight } y &\rightarrow 5 :: Z \\
\text{dist}(x, y) &\rightarrow 4 :: Z
\end{align*}
\]

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 \( a : u \rightarrow x \) (called coding function) is one-to-one. Let us write \( C \subseteq T^n \) to denote the image of \( a \).
- 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 \( b : y \rightarrow x \) (called decoding function) of a subset \( D \subseteq T^n \) onto \( C \).

If \( y \in D \), \( y \) 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 \( a(u') = x \) as the decoded block.

Here is now the key definition: If \( t \) is a positive integer, we say that \( b \) has correcting capability \( t \) if \( y \in D \) and \( x' = x \) (hence also \( u' = u \)) whenever \( d(x, y) \leq t \). 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 \( x,y \in C, x \neq y \) (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 \( d+k \leq n+1 \) (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.

---

**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 \( G \in M_k^n(F) \) 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 \( F^k \rightarrow F^n \) such that

\[ u \rightarrow x = uG \]

as a coding function. Since

\[ uG = u_1G^1 + \cdots + u_nG^n, \]

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 \( \Omega \)-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 \([7,4,3]\)**

Let \( K \) be the field of binary digits.

Let \( K = \mathbb{Z}_2 \)

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

\[ R = \begin{bmatrix} u, 0, 1, 1 \\ 1, 1, 0, 1 \\ 0, 1, 1, 1 \end{bmatrix} \]

(\( u \) being in \( K \), all other entries are projected to \( K \))

The generator matrix of \( C \) is the transpose of \( R \) and the identity matrix \( \text{Id}(4) \)

\[ G = (R' \& \text{Id}(4))' \]

The control matrix of \( C \) is the transpose of \( R \) and the identity matrix \( \text{Id}(3) \)

\[ H = \text{Id}(3) \& R' \]

**Encoder**

\[ \text{hamming_encoder}(u) := u*G \]

(Note that \( u*G = uR' \mid u \))

**Decoder**

\[ \text{hamming_decoder}(y) := \]

begin
\begin{verbatim}
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») # position of s in H
    x=y
    j=index(H,s)
    show(«Error in position »,j)
else print(«Non-decodable vector»)
end
\end{verbatim}

---
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:

\[
H = \begin{bmatrix} h_1 & \cdots & h_n \\ h_1 \alpha_1 & \cdots & h_n \alpha_n \\ \vdots & \ddots & \vdots \\ h_1 \alpha_1^{r-1} & \cdots & h_n \alpha_n^{r-1} \end{bmatrix}
\]

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 \( \alpha_i \uparrow \alpha_i \) for all \( i \uparrow i \). We will refer to \( h = [h_1, \ldots, h_n] \) and \( \alpha = [\alpha_1, \ldots, \alpha_n] \) as the vector \( h \) and the vector \( \alpha \) of the control matrix. Here is an \( \Omega \)-function that constructs \( H \):

\[
ACM(h,a,r) :=
\begin{align*}
\text{begin local } H, \# \text{ the matrix } \\
f \# \text{ current vector of } H \\
f=h; H=[f] \\
\text{for } i \text{ in } 1..(r-1) \text{ do} \\
\quad \text{f=prod(f,a); } H = H \# f \\
\text{end}
\end{align*}
\]

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

\[
\text{prod}(a,b) := [a.i*b.i \text{ for } i \text{ in range(a)}]
\]

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 \( \alpha \), so it is convenient to have precomputed the vector \( \beta \) of these inverses. The more convenient way for us to achieve this here is to assign the five objects \( h, \alpha, \beta, H, r \) 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).

\[
\begin{align*}
\# & \text{ To construct alternant codes} \\
\text{Alternant}(h,a,r) := \\
\text{begin} \\
\quad vh=h; va=a; vb=inv(a); H=ACM(h,a,r); cd=r \\
\text{end}
\end{align*}
\]

This listing illustrates the function overloading capacity of OMEGA, as the call \( \text{Alternant}(a,r) \) is defined in terms of the more general call \( \text{Alternant}(h,a,r) \). 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,

\[
\begin{align*}
\text{f(x:Rational)} & := 1 \\
\text{f(x)} & := 0
\end{align*}
\]

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.

\[
\begin{align*}
\# & \text{ Vandermonde matrix of codimension } r \text{ on the} \\
\text{vector } a \\
\text{vandermonde(a,r)} & := \text{ACM(constantvector(1, dim(a)), a, r)'} \\
\# & \text{ Vandermonde square matrix on the vector } a \\
\text{vandermonde(a)} & := \text{ACM(constantvector(1, dim(a)), a, dim(a))'}
\end{align*}
\]
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 designated 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 \( h = [1,a,a^2,\ldots,a^{n-1}] \) 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 \( \text{series}(a,n) \) that returns \([1,a,a^2,\ldots,a^n]\):

\[
\text{series}(a,n) := [a^i \text{ with } i \text{ in } 0..(n-1)]
\]

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

\[
\text{BCH}(a,d,l) := \text{BCH}(a,d,1)
\]

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:

\[
\text{RS code of codim } r \text{ associated to the field } K \text{ for } K' = K
\]

\[
\text{RS}(K,r) := \text{BCH}(\text{prim}(K),r+1)
\]

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 \( \alpha \) 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 \( g(\alpha) \uparrow 0 \) 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 \( \alpha \), vector \( \alpha \) equal to \( \alpha \), and with order \( r \) equal to the degree of \( g \). So \( \Omega \) can be used as follows:

\[
\# \text{ Classical Goppa codes}
\]

\[
\text{Goppa}(g,a) := \text{Alternate}(\text{inv(eval}(g,a)),a,\text{deg}(g))
\]

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 \( \alpha \), that is, \( \beta_1 = \alpha_1^{-1} \).

The BM decoding algorithm
This algorithm takes as input a vector \( y \in K^\ell \) and it outputs, if \( y \) turns out to be decodable, a code vector \( x \).

1. Find the syndrome \( s = yH \), say \( s = [s_0,s_1,\ldots,s_{\ell-1}] \).
2. Transform \( s \) into a polynomial \( S \) in the indeterminate \( z \):

\[
S = s_0 + s_1z + \ldots + s_{\ell-1}z^{\ell-1}
\]

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

\[
r_{i+1} = q_i r_i + r_{i+1},
\]

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

\[
\psi_0 = v_0,v_1 = v_1 + q_1 v_j \]

such that

\[
\psi_{j+1} = \psi_j + q_j v_j
\]

(note that \( r_{j+1} = r_j + q_j r_j \)). We set \( \sigma = v_\ell \) (this is called the error locating polynomial) and \( e = r_\ell \) (this is called error resolving polynomial).
4. Make a list \( L = \{m_1,\ldots,m_{\ell}\} \) of the indices \( j \in L = \{1,\ldots,n\} \) such that \( \sigma(e) = 0 \). These indices are called error locations. If \( s \) is less than the degree of \( \sigma \), 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

\[
y_j + \frac{\alpha_j \cdot e(\beta_j)}{\sigma'(\beta_j)}
\]

where \( \sigma' \) is the derivative of \( \sigma \).

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 \( v_1 \) and \( r_1 \), 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 \( a = [a_1, \ldots, a_n] \), the indices \( j \) in \( \{1, \ldots, n\} \) such that \( a_j \) is a zero of \( f \).

code:

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 \( v_h, v_a \) and \( v_b \), holding vectors of the same length \( n \), with \( v_b \) the component-wise inverse of \( v_a \), 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 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 \( L \):

\[
\text{flip} (y,L):=\text{for } j \text{ in } L \text{ do } y.j=1-y.j \text{ end}
\]

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 \( d \geq r+1 \) (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 \( r \) that can correct \( t \) errors. What is the minimum possible \( q \) we have to take? The answer is easy, as we are assuming that \( k \geq pn \) and \( n = q-1 \), so that \( q = n+1 = k+d \geq pq-p+2t+1 \), or

\[
(1-p)q \geq 2t+(1-p).
\]

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

\[
q \geq 1+2tr.
\]

Let us write an \( \Omega \)-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 \( x \)

\[
\text{next}_q(x) := \begin{array}{ll}
\text{begin local } q \\
q = \text{ceil}(x) \\
\text{while primepower?}(q) = \text{false} \text{ do} \\
q = q + 1 \\
\text{end} \\
q 
\end{array}
\]

# To find the least number \( q \) that is a prime power which is equal or greater than \( 1+2t/r \), \( t \) and \( r \) given

\[
\text{next}_q(r, t) := \text{next}_q(1+2t/r)
\]

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:

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

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:

\[
\begin{align*}
\text{Z3} &= \mathbb{Z}_3 \\
\text{clear } x \\
f &= \text{polirred}(\text{Z3}, 3, x) \\
\text{K} &= \text{ext}(\text{Z3}, f) \\
k &= \text{card}(\text{K}) - 1; r = 10 \\
\text{RS}(\text{K}, r)
\end{align*}
\]

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 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,..., \( n \)\}:

\[
\text{rd_comb}(n, m) := \\
\text{begin local } c, x \\
c = (\text{null}) \\
\text{while } \text{dim}(c) < m \text{ do}
\]

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

\[
\text{rd}(K) := \text{elem}(\text{random}(1, q-1), K)
\]

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

\[
\text{rd}(s, K) := (\text{rd}(K) \text{ with } i \text{ in } 1..s)
\]

Random error patterns. Given the length of the code, \( n \), a weight \( s \), \( 0 \leq s \leq n \), 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:

\[
\text{rd_err_vec}(s, K, n) := \\
\text{begin local } E, L, e, i \\
E = \text{rd}(s, K); L = \text{rd_comb}(n, s) \\
e = \text{constantvector}(0, n) \\
\text{for } i \text{ in range}(E) \text{ do} \\
e = e + (E.i) \times \text{eps}(L.i, n) \\
\text{end}
\]

If we do not specify \( n \), let its default be \( n = \text{card}(K)-1 \):

\[
\text{rd_err_vec}(s, K) := \text{err_vec}(s, K, \text{card}(K)-1)
\]

Finding code vectors. If \( a \) is the primitive element of \( K \) with which we constructed the control matrix \( H \), then \( h = [1, a, a^2, \ldots, a^{n-1}] \) and \( a = h \), and so the columns of \( H \) have the form \([1, a, a^2, \ldots, a^{n-1}]\), \( i = 1, \ldots, r \) (the Vandermonde matrix of \( n \) rows on the elements \( a, a^2, \ldots, a^{n-1} \)). 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 \( x_i \) of the form \( x_i = [1, a^i, a^{2i}, \ldots, a^{(n-1)i}] \) are code vectors for \( j = 0, \ldots, k-1 \). Actually the vectors \( x_0, \ldots, x_{k-1} \) are linearly independent (note that the corresponding matrix is the Vandermonde matrix of \( k \) rows on \( 1, a, a^2, \ldots, a^{k-1} \)) and so form a basis of the code. Here is a direct construction of this generator matrix:

\[
\text{rs}_G(a, n, k) := \\
[\text{series}(a^j, n) \text{ with } j \text{ in } 0..(k-1)]
\]

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:

\[
\text{rd_lin_comb}(G) := \\
\text{begin k, n, t} \\
k = \text{dim}(G); n = \text{dim}(G.1) \\
t = [\text{elem}(\text{random}(n+1), K) \text{ with } i \text{ in } 1..k] \\
t \times \text{G} \\
\text{end}
\]
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 $\text{bm}(y)$, we will return $x-\text{bm}(x+e)$, which should be zero if $s$ is not greater than the error-correcting capacity $r/2$. Here is the corresponding $W$-function:

```wolfram
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)})
end
```

Now it will be enough to evaluate expressions of the form $\text{rs decoder trial}(s)$ 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) # beyond the correcting capacity
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+t+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+2t+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)}
```

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)
```

Example: RS[80,60,21]
If, instead of $q = 81$, we take $q = 83$, and still $r = 20$, we get an $\text{RS}[82,62,21]$ that performs very similarly, with a slightly better rate, but with the faster arithmetic mod 83 than that of $\mathbb{F}_{81}$.

**Example:** $\text{RS}[120,96,25]$

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
```

6. Some auxiliary and complementary packages

In this chapter we are going to look at several W-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)}
```

**Example:**
```
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)}
```

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 = vi.

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
(0) d=distribution(v)
(0) c={domain(d)} # the list of keys of table d
(0) m=nops(d(c.1)) # number of occurrences of c.1
(0) for i in 2..dim(c) do
(0) if (k=nops(d(c.i)))>m then m=k end
(0) end
(0) return C
end
```

# 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.omm. # Alternatively, write the expression read # ‘we:/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
# 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 \(\text{brack}(a,x)\) (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 \(1,x,º, xr-1\) of \(K[x]\) over \(K\) (so \(r\) is the degree of the extension \(K[x]/K\)). It uses the internal function \(\text{components}(a)\), which gives the sequence of these components when \(a\) is not in \(K\).

\[
\text{brack}(a,x):=
begin
\text{if \(\text{subfield?}(\text{field}(a),\text{precedent}_\text{ext}(\text{field}(x)))\) then } \{a\} \mid \text{constantlist}(0, \text{relativedeg}(x))-1\) \text{ else } \{\text{components}(a)\}
end
end
\]

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

\[
Z_p = K_0 \subset K_1 \subset \ldots \subset K_n \subset K_p = K
\]

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

\[
\text{subfield?}(K,F):=
begin
\text{local } q
q=\text{card}(K)
\text{if } \text{car}(K)! = \text{car}(F) \text{ | } q=\text{card}(F) \text{ then } \text{return } \text{false end}
\text{while } \text{card}(F)>q \text{ do } F=\text{precedent}_\text{ext}(F) \text{ end}
end
\]

if \(F=K\) then true
else false end
end

Now the function \(\text{prima}(h:\text{VEC},x)\) takes the vector \(h\) of length \(n\) over \(K[x]\) and makes a vector of length \(nr\) over \(K\) with the brack of the components of \(h\), and the function \(\text{prima}(H:\text{MAT},x)\) takes the matrix \(H\) of type \(m\times n\) over \(K[x]\) and makes a matrix of type \(mr\times nr\) over \(K\) by successively applying \(\text{prima}\) to the rows of \(H\).

\[
\text{prima}(H:\text{MAT},x):=
begin\local h1, i
H1=null
for h in H do H1=(H1,\text{prima}(h,x)) end
[H1]
end
\]

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

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

\[
\text{load}\ «e:/\text{OMEGA}/\text{tools}\»
\]
\[
\text{Z2=Zn}\ 2
\]
\[
clear x
\]
\[
F=\text{ext}(\text{Z2, }x^3+x+1)
\]
\[
n=\text{card}(F)
\]
\[
f8=[0]\mid \text{series}(x,n-1) \quad \# \text{the set of elements of } F
\]
\[
---->
[0, 1, x, x^2, (x+1), (x^2+x), (x^2+x+1), (x^2+1)] :: \text{Vector}(\text{F}(2^3))
\]
\[
g=T^2+T+1
\]
\[
h=[\text{eval}(g,x) \mid x \in f8]
\]
\[
---->
[1, 1, (x^2+x+1), (x+1), (x^2+1), (x^2+1), (x+1)] :: \text{Vector}(\text{F}(2^3))
\]
\[
H= (h \mid \text{prod}(h,f8))'
\]
\[
---->
[1, 0]
[1, 1]
[\{x^2+x+1\}, (x^2+1)]
[\{x+1\}, (x^2+x+1)\}
[\{x^2+x+1\}, x]
[\{x^2+1\}, (x+1)]
[\{x^2+1\}, (x^2+x)]
[(x+1), x^2]
:: \text{Matrix}(\text{F}(2^3))
\]
\[
H1=\text{prima}(H,x)
\]
\[
---->
\]

\[
212 \text{ S. Xambó}
\]
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 $\text{vol}(n, r, q)$, and $\text{vol}(n, r)$ when $q = 2$. We can define them by translating to Omega the usual formulae:

$$\text{vol}(n, r, q):= \sum \text{binomial}(n, i) \cdot (q-1)^i \text{ with } i \text{ in } 0..r$$

$$\text{vol}(n, r):= \sum \text{binomial}(n, i) \text{ with } i \text{ in } 0..r$$

$$\text{vol}(23, 3)\quad \rightarrow \quad 2048$$

$$\text{vol}(11, 2, 3)\quad \rightarrow \quad 243$$

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

$$\text{perfect}(C, q):= q^{-(C.1-C.2)} = \text{vol}(C.1, (C.3-1)/2, q)$$

$$\text{perfect}(C):= 2^{-(C.1-C.2)} = \text{vol}(C.1, (C.3-1)/2)$$

$$\text{perfect}([23, 12, 7])\quad \# \text{ the Golay binary code}$$

$$\text{perfect}([11, 6, 5], 3)\quad \# \text{ the Golay ternary code}$$

$$\rightarrow \quad \text{true}$$

More generally, we can introduce the notion of perfection of a code as the quotient of $q^n \text{vol}(n, t, q)$, 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).

$$\text{perfection}(C, q):= \text{vol}(C.1, (C.3-1)/2, q) / q^{-(C.1-C.2)}$$

$$\text{perfection}(C):= \text{perfection}(C, 2)$$

$$\text{perfection}([11, 6, 5], 3)\quad \rightarrow \quad 1 \quad : \quad Z$$

$$\text{perfection}([32, 6, 15])\quad \rightarrow \quad 0.067$$

Bound on code parameter’s and related functions

There are many bounds other than the Singleton upper bound $k \leq n+1-d$, valid under diverse circumstances (cf. [19]), for the dimension of a code.

In this section, we collect a number of Omega-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 Omega functions, $\text{gilbert}$ and $\text{gilbert_varshamov}$, are just a straightforward translation of the usual formulae and procedures:

$$\text{lb_gilbert}(n, d, q) := \text{ceil}(q^n/\text{vol}(n, d-1, q))$$

$$\text{lb_gilbert}(n, d) := \text{ceil}(2^n/\text{vol}(n, d-1))$$

$$\text{lb_gilbert}(10, 3)\quad \rightarrow \quad 19$$

$$\text{lb_gilbert}(11, 3)\quad \rightarrow \quad 31$$

$$\text{lb_gilbert_varshamov}(n, d, q):= \begin{cases} k=0 & \\
\text{while } q^{-(n-k)} > \text{vol}(n-1, d-2, q) \text{ do } k=k+1 \text{ end} \\
k-1 & \end{cases}$$

$$\text{lb_gilbert_varshamov}(n, d) := \text{lb_gilbert_varshamov}(n, d, 2)$$

$$\text{lb_gilbert_varshamov}(10, 3)\quad \rightarrow \quad 6$$

$$\text{lb_gilbert_varshamov}(10, 3)\quad \rightarrow \quad 6$$

$$\# \text{ (compare the value } 2^6=64 \text{ with }\text{ lb_gilbert}(10, 3) \rightarrow \quad 19)$$

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:

$$\text{ub_sphere}(n, d, q) := \text{floor}(q^n/\text{vol}(n, (d-1)/2, q))$$

$$\text{ub_sphere}(n, d) := \text{floor}(2^n/\text{vol}(n, (d-1)/2))$$

$$\text{ub_sphere}(10, 3)\quad \rightarrow \quad 93$$

$$\text{ub_sphere}(11, 3)\quad \rightarrow \quad 170$$

For binary codes, this bound is improved by the Johnson bound. We introduce the function $\text{ub_johnson}$, 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
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
ub_johnson(13,5)
--- 77

Now we list the Omega 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(13,5)
--- 6
ub_griesmer(14,9,3)
--- 4

The Omega 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):= 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-\sum_{j=0}^{t}p^j(1-p)^n. Taking into account the optimization in the calculation of some intermediate expressions, the resulting Omega-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 Omega 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 = (d-1)/2, 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)

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 \( \text{ord}(a) \) that gives the order of a non-zero element of a finite field), here is a presentation as an external \( \Omega \)-function that uses fast modular arithmetic:

\[
\text{ord}(q,n) := \\
\text{begin local } d, \ A, u, i, r \in \mathbb{Z}_n \text{ for the unit of } A, \phi(n) \text{ for the } \phi \text{-function of } n \text{, and } \text{divisors } \text{of } n \text{ as lists of divisors of } \phi(n)\; \\
\text{if } \gcd(q,n) > 1 \text{ then return } \text{show}(\text{The number } q, \text{ is not invertible mod } n) \text{ end else } \; \\
\ d = \text{set}(\text{divisors} \phi(n)) \; \text{A} = \mathbb{Z}_n \qquad q = q:A; u = 1:A \; \\
i = 1; r = d.i \; \text{while } q^r \neq u \text{ do } \; i = i + 1; r = d.i \; \\
\text{end end} 
\]

For example, the expression

\[
n = 571725 \\
\{\text{ord}(q,n) \text{ with } q \text{ in } 2..100 \text{ where } \gcd(q,n) = 1\}
\]

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

Cyclotomic classes

In the factorization of \( X^n - 1 \) over \( \mathbb{F}_q \), and assuming \( \gcd(n,q) = 1 \), the irreducible factors are in one-to-one correspondence with the \( q \)-cyclotomic classes of \( \mathbb{Z}_n \), where the \( q \)-cyclotomic class of \( j \in \mathbb{Z}_n \) is \( \{j, jq, jq^2, \ldots\} \) (the products computed mod \( n \)). The factor corresponding to the class \( C \) is the polynomial \( f_C = \Pi_{\omega \in C} (X - \omega) \), where \( \omega \) is a primitive \( n \)-th root of unity in a field containing \( K \).

\[
\text{cyclotomic_class}(j,n,q) := \\
\text{begin local } C, \ k \text{ if } \gcd(n,q) > 1 \text{ then return } \text{show}(\text{The number } q, \text{ is not invertible mod } n) \text{ end else } \; \\
\ j = j \mod n; C = \{j\} \; \text{K} = (j^q) \mod n \text{ while } k \neq j \text{ do } \; \text{C} = \text{C | (K)} \text{ k = (K^q) \mod n} \text{ end} \; \text{C} \end{array}
\]

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

\[
\text{cyclotomic_classes}(n,q) := \\
\text{begin local } \ J, C, c; \text{ if } \gcd(n,q) > 1 \text{ then return } \text{show}(\text{The number } q, \text{ is not prime to } n) \text{ end else } \; \\
\ J = \{0\}; C = \{0\} \text{ for } j \text{ in } 1..(n-1) \text{ do } \; \text{if index}(J,j) = 0 \text{ then } \; \\
\ c = \text{cyclotomic_class}(j,n,q) \; \text{C} = \{C,c\}; J = J | c \text{ end end} \; \\
\text{end end} 
\]

\[
\text{cyclotomic_classes}(n,q) := \text{cyclotomic_classes}(n,2) 
\]

\[
\text{cyclotomic_classes}(15) \rightarrow \{(0), (1,11), (2,7), (3), (4,14), (5,10), (6), (8,13), (9), (12)\}
\]

Since the class of 0 is \( \{0\} \), and the corresponding factor is \( X-1 \), we see that \( X^{15}-1 \) factors over \( \mathbb{Z}_2 \) 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 \( \mathbb{Z}_{11} \), into 5 linear factors and 5 quadratic factors.

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

\[
\text{omega}(K: \text{Field, n:Integer}) := \\
\text{begin local } q, r, f, t, F, a \text{ if } r > 1 \text{ then } \; \\
\ f = \text{polirred}(K,r,t) \text{ F = ext}(K,t,f) \; \text{take } a = \text{prim}(F) \text{ a^((q^r-1)/n) end end} 
\]

(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 \( X : K^* \rightarrow \{-1,1\} \) 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 \( \Omega \)-function that gives this character:

\[
\text{legendre}(a) := \text{if } a = 0 \text{ then 0 end} 
\]
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 $X (X^T X)$. We can instruct OMEGA to obtain this expression as follows:

```python
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)

```plaintext
[ 0, 1,-1,-1, 1]
[ 1, 0, 1,-1,-1]
[-1, 1, 0,-1, 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 $\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$, 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 $H_1 \cdot H = I$ and note the compact expression for this matrix in the function body.

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

hadamard(n: Integer) check n>0 :=
begin
  local q, u, S
  q := card(K)
  u := constantvector(1, q)
  S := idmatrix(q) + paley_matrix(K)
  ( \begin{pmatrix} 1 \\ u \end{pmatrix} & \begin{pmatrix} u & -S \end{pmatrix} 
end
```

hadamard(Zn 3)

```plaintext
[1, 1, 1, 1]
[1,-1, 1,-1]
[1,-1,-1, 1]
[1, 1,-1,-1]
: Matrix(Z)
```

It is to be remarked that if $S$ is a matrix of type $m \times n$ and $u$ a vector of dimension $m$, then $u \mid S$ yields the matrix of type $m \times (n+1)$ obtained by attaching the column vector $u'$ to the left of $S$. Likewise, if $u$ is a vector of dimension $n$, then $S \mid u$ yields the matrix of type $m \times (n+1)$ 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 $F_q$, 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 $k \times n$ generating matrix $G$ has the form

$\begin{pmatrix} 0, \ldots, 0, g_0, \ldots, g_r, 0, \ldots, 0 \end{pmatrix}$

with $i-1$ leading zeros. Similarly, if $h = (X^n)^{i/2}$, and $h = h_0X^{n-i}+\ldots+h_nX+h_0$, then the $i$-th row of a control matrix $H$ that is dual to $H$ has the form

$\begin{pmatrix} 0, \ldots, 0, h_0, \ldots, h_n, 0, \ldots, 0 \end{pmatrix}$

again with $i-1$ leading zeros.

The next three functions allow us to construct these matrices.

```python
cyclic(g: VEC, n):=
begin
  local r, k, G
  r := dim(g)-1; k := n-r
  if k>1 then g := g \mid constantvector(0, k-1) end
  G := g
  for i in 1..(k-1) do
```
\[
\begin{align*}
g &= [0] \mid \text{take}(g, n-1) \\
G &= (G, g) \\
\text{end}
\end{align*}
\]

These functions do not check whether \( g \) is a divisor of \( X^n-1 \), nor care about the base field \( \mathbb{F}_q \). The point is that we will first obtain \( g \), for example by calling \( \text{factor}(X^n-1, K) \) (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:

\[
\begin{align*}
\text{resultant\_matrix}(f: \text{VEC}, g: \text{VEC}) &:= \\text{cyclic}(f, \text{dim}(f)+\text{dim}(g)-2) \\
&\quad \& \\text{cyclic}(g, \text{dim}(f)+\text{dim}(g)-2)
\end{align*}
\]

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

\[
\begin{align*}
\text{pol2vector}(P, T) &:= \\
\text{begin local c, C, k} \\
c &= \text{subst}(P, T, 0) \\
C &= [c] \\
P &= \text{diff}(P, T) \\
k &= 1 \\
\text{while P} \neq 0 \text{ do} \\
c &= \text{subst}(P, T, 0) \\
\text{if c} \neq 0 \text{ then c} &= c/k! \text{ end} \\
C &= C \mid [c] \\
P &= \text{diff}(P, T) \\
k &= k+1 \\
\text{end} \\
C &\text{ end}
\end{align*}
\]

Now the question can be solved as follows:

\[
\begin{align*}
\text{resultant\_matrix}(f, g, T) &:= \\
\text{resultant\_matrix}(\text{pol2vector}(f, T), \text{pol2vector}(g, T))
\end{align*}
\]

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

\[
\begin{align*}
\text{discriminant\_matrix}(f, T) &:= \\
\text{resultant\_matrix}(f, \text{diff}(f, T), T)
\end{align*}
\]

Let us illustrate how the above functions work in a short session (cf. [3], p. 233-4):
For example, the binary Golay code can be defined as the cyclic code of length \( n = 23 \) generated by
\[
g = x^{11}+x^9+x^7+x^6+x^5+x+1 \in \mathbb{Z}_2[x]
\]
and, since in this case the error-correcting capacity is 3, the Meggitt table can be defined as follows:
\[
E = \left[ \text{rem}(x^{n-1}, g) \rightarrow x^{n-1}(n-1) \right] + \\
\left[ \text{rem}(x^{n-1}+x^i, g) \rightarrow x^{n-1}+x^i \text{ with } i \text{ in } 0..(n-2) \right] + \\
\left[ \text{rem}(x^{n-1}+x^i+x^j, g) \rightarrow x^{n-1}+x^i+x^j \text{ with } (i,j) \text{ in } 0..(n-2), 0..(n-2) \text{ where } j<i \right]
\]

Actually the type of this sort of table (with brackets instead of braces) is \( \text{Divisor} \). 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 \( i = 0, \ldots, 21 \), or that of \( x^{22}+x^{i}+x^{j} \) for \( i, j \in \{0,1,\ldots,21\} \) and \( i > j \). Otherwise \( E(s) \) selects, among those polynomials, the one that has syndrome \( s \).

Now we implement the Meggitt decoder as an \( \Omega \)-function \( \text{meggitt}(y,g,n) \) 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 \( j = 1,2,\ldots,n-1 \), the syndromes \( s_j \) of \( xy \), and stop for the first \( j \) such that \( e = E(s) \neq 0 \).
- 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 \( y \).

\[
\text{meggitt}(y,g,n) := \\
\text{begin local } x, s, j \\
x = \text{var}(g) \quad \# \text{ the variable of } g \\
s = \text{rem}(y,g) \quad \# \text{ the syndrome} \\
\text{if } s == 0 \text{ then return} \\
\text{show("Word code: ",y) end} \\
\text{j = 0} \\
\text{while } E(s) == 0 \text{ do} \\
\text{j = j+1} \\
\text{s = rem(x^j*y,g)} \\
\text{end} \\
y = y - \text{lcoef}(E(s))*(n-1-j) \\
\text{show("Error in degree ", n-1-j, " is corrected")} \\
\text{meggitt}(y,g,n) \\
\text{end}
\]

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

\[
\text{Z}_2= \mathbb{Z}_2; \ u=1:Z_2 \\
n=23 \\
clear x \\
g=x^{11}+x^9+x^7+x^6+x^5+x+u \quad \# \text{ generator of } \text{Golay2} \\
\# \text{ Construction of Meggitt table } E. \text{ We do not} \\
\# \text{ list it again: see the first listing} \\
\# \text{ in the last section} \\
y = x^{20}+x^{15}+x^{10}+x^5+u \\
\text{meggitt}(y,g,n) \\
\text{--->} \\
\text{Error in degree 14 is corrected} \\
\text{Error in degree 12 is corrected} \\
\text{Error in degree 4 is corrected} \\
\text{Code word:} \\
x^{19}+x^{15}+x^{14}+x^{12}+x^{10}+x^5+x^4+1 :: \text{Z}_2[x]
\]

The ternary Golay code can be defined as the cyclic code of length 11 generated by
\[
g = x^5+x^4+2x^3+x^2+2 \in \mathbb{Z}_3[x]
\]
and in this case, since the error-correcting capacity is 2, the Meggitt table can be defined as follows (with \( n = 11 \) and \( \mathbb{Z}_3 = \mathbb{Z}_3\{0\} \)):
\[
E = \left[ \text{rem}(a^*x^{n-1}, g) \rightarrow x^{n-1}(n-1) \text{ with } a \in \mathbb{Z}_3 \right] + \\
\left[ \text{rem}(a^*x^{n-1}+b^*x^i, g) \rightarrow a^*x^{n-1}+b^*x^i \text{ with } (i,a,b) \text{ in } 0..(n-2), \mathbb{Z}_3, \mathbb{Z}_3 \right]
\]

Now we can work out examples like the following:

\[
\text{Z}_3= \mathbb{Z}_3; \ u=1:Z_3 \\
n=11 \\
z_3=[u,2] \\
clear x \\
g=x^5+x^4+2x^3+x^2+2*u \\
\# \text{ Construction of Meggitt table, as explained} \\
\# \text{ above. We do not repeat it here.} \\
y = x^5+x^4+2*x+u \\
\text{meggitt}(y,g,n) \\
\text{--->} \\
\text{Error in degree 9 is corrected} \\
\text{Error in degree 5 is corrected} \\
\text{Code word:} \\
x^9+x^5+x^4+x^3+x+1 :: \text{Z}_3[x]
\]

The factors to construct the Golay codes can be obtained by the internal function \( \text{factor}(f,K) \):
\[
\text{factor}(x^{23}-1,\mathbb{Z}_2) \\
\text{--->}
\]

218 S. Xambó
\begin{verbatim}
{{x+1,1}, {x^5+x^4+2x^3+x^2+2,1}},
{x^5+2x^3+x^2+2x+2,1}}
\end{verbatim}

\texttt{factor(x^11-1,Zn 3)\rightarrow}
\begin{verbatim}
{{x+2,1}, {x^11+x^9+x^7+x^6+x^5+x+1,1}},
{x^11+x^10+x^9+x^6+x^5+x^4+x+2,1}}
\end{verbatim}

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 \texttt{macwilliams} returns the weight enumerator of the dual code:

\begin{verbatim}
macwilliams(A,n,k,q):=
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
\end{verbatim}

For example, the expression

\begin{verbatim}
Phi(4725,T)\rightarrow
\end{verbatim}

\begin{verbatim}
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]
\end{verbatim}
is evaluated in 0.06 s, and the expression

\begin{verbatim}
{Phi(n,T) with n in 1..100 where
not(prime?(n))}
\end{verbatim}
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 \( t = [T] \mod f \) 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 \textit{order of} \( f \), and \( f \) is said to be \textit{primitive} if and only if \( t \) is a primitive element of \( F \). The order of \( f \) can thus be calculated by the following function:

\begin{verbatim}
ord(p,K):=
begin local x, f, t, F
x=var(p)
f=subst(p,x,t)
F=ext(K,f)
ord(t)
end
\end{verbatim}

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 \( K \).

\begin{verbatim}
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
\end{verbatim}

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 \texttt{listirred(K,r,T)}, which gives the list of all the monic irreducible polynomials of degree \( r \) over \( K \), and the function \texttt{ord(p,K)} explained above:

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

These functions are reasonably efficient. For example, to find one irreducible primitive polynomial of degree \( i \) over \( Z_2 \), 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 \( Z_2 \) (the
number of irreducible polynomials of degree 15 over $\mathbb{Z}_2$ is 2182).

The other approach, namely factoring $F(q-1, T)$ 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

$$(\text{see [20], §1.2}).$$

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.

\begin{align*}
\text{newton}(x,j) &:= (\text{product} (x-i+1) \text{with} i \in 1..j)/j! \\
\text{Kr}(x,k,n,q) &:= \sigma (-1)^j \text{newton}(x,j) \text{newton}(n-x,k-j) (q-1)^{k-j} \text{with} j \in 0..k \\
\text{Kr}(X,0,n,q) &\longrightarrow 1 :: \mathbb{Z} \\
\text{Kr}(X,1,n,q) &\longrightarrow -Xq+nq-n :: \mathbb{Z}[X,n,q] \\
\text{Kr}(X,2,n,q) &\longrightarrow -1/2(X^2q^2 + X(-2nq+2n+q-2)q + n^2q^2-2n^2q+n^2-nq^2+2nq-n) :: \mathbb{Q}[X,n,q] \\
\text{Kr}(X,3,n,q) &\longrightarrow -1/6X^3q^3 + 1/2X^2(nq-n-q+2)q^2 - ... :: \mathbb{Q}[X,n,q]
\end{align*}

Let us check the orthogonality relation

\begin{align*}
\sum_{i=0}^{n} \text{Kr}(i)\text{Kr}(k) = \delta_{i\neq k} q^i .
\end{align*}

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

$n = 10$

$W = \{ \text{subst}(\text{Kr}(x,i,n),x,k) \text{with} i \in 0..n\} \text{with} k \in 0..n\}$

\begin{align*}
\text{V} &\longrightarrow 1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1 \\
\text{Kr}(x,0,n,q) &\longrightarrow 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 \quad \text{::} \mathbb{Z} \\
\text{Kr}(x,1,n,q) &\longrightarrow 1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1 \quad \text{::} \mathbb{Z} [x,n,q] \\
\text{Kr}(x,2,n,q) &\longrightarrow 2, 11, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1 \quad \text{::} \mathbb{Q}[x,n,q] \\
\text{Kr}(x,3,n,q) &\longrightarrow 6, 26, 27, 13, 13, 27, 45, 210, 42, -14, -14, 42, 210, 252, 0, -28, 0, 12, 0, -12, 0, 28, 0, -252 \quad \text{::} \mathbb{Z}[x,n,q]
\end{align*}

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 $\mathbb{Z}_2$.

\begin{align*}
&Z_2 = \mathbb{Z}_n 2 \\
f(x,y) := x^3y+y^3+x \\
&\text{clear} x \\
&F8=\text{ext}(Z_2,x^3+x+1) \\
&n=\text{card}(F8)-1 \\
f8 = [0] \mid [x^i \text{ with} i \in 0..n-1] \\
X = \{ [x,y] \text{ where} f(x,y)=0 \} \text{ with} (x,y) \text{ en} \{f8,f8\}$
\end{align*}

\begin{align*}
&\longrightarrow [1024, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] \\
&\text{V^W'} \\
&\longrightarrow [120, -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]
\end{align*}
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 $\mathbb{Z}_2$ (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

$\mathbb{Z}_2 = \mathbb{Z}/2$
clear $t$

$F_{64} = \text{ext}(\mathbb{Z}_2, t^6+t+1)$

clear $x$

$F = \text{ext}(\mathbb{F}(2^3), x^4+x^3+1)$

$P = \{\{a,b\} \mid \text{g}(a,b)=0\}$

$\text{dim}(P) \rightarrow 512 :: \mathbb{Z}$

$\text{Y}.100 \rightarrow \{t^5+t+1, t^5+t^3+1\} :: \text{Vector}(F(2^6))$

Rational functions

Among the fundamental services needed for computational work in the field of algebraic-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 $f = (u^2v+uv^3+2)/(u^15+uv^7-7)$ can be evaluated at the point $[2,3]$ as follows:

clear $u, v$

$f = (u^2*v+u*v^3+2)/(u^15+u*v^7-7)$

$P \rightarrow [2,3]$

eval(f, P)

$\rightarrow 68/37135 :: \mathbb{Q}$

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 $\Omega$-expression

$$[[\text{eval}(f,i,P,j) \mid j \in \text{range}(P)] \mid i \in \text{range}(f)]$$

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.

Two cornerstone 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 $-c$. Let us define an $\Omega$-function for producing these dual classes:

dual(c) := $[(-1)^i * c_i \mid i \in \text{range}(c)]$

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 $u_1v_1+\cdots+u_kv_k$, with the convention that $u_i$ (v_j) is taken to be zero if $i > \text{dim}(u)$ ($j > \text{dim}(v)$), or if $i < 0$ ($j < 0$).

convolution($u, v, k$) := $\sigma u_i * v_{(k-i)}$

where $i < \text{dim}(u)$ and $k-i < \text{dim}(v)$ with $i$ in $1..(k-1)$

The two functions to which this section is devoted are $c_2p$ and $p_2c$, which transform, respectively, the total Chern class $c = [c_1, c_2, \ldots, c_n]$ into the total Chern character $p = [p_1, p_2, \ldots, p_n,n!]$ and conversely. We just apply the formulae on p. 56 of [5]. In any case, these functions are cornerstones of the $\Omega$-library for intersection theory computations (see [25]), like the Maple functions $\text{expp}$ and $\text{logg}$ in the Schubert package of S. Katz and S. A. St"{o}rme.

c2p($c, r: \text{Integer}$) :=
begin
local $p$, $sk$
\[ p = [c.1] \]
\[ c = \text{dual}(c) \]
\[ \text{for } k \text{ in } 2..r \text{ do} \]
\[ sk = \text{convolution}(c, p, k) \]
\[ sk = -(k*c.k + sk) \]
\[ p = p \mid [sk] \]
\[ \text{end} \]
\[ \{p, k/k! \text{ with } i \text{ in range}(p)\} \]
\[ \text{end} \]

\[ p_2c(p, r) := \]
\begin{verbatim}
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{verbatim}

\[ c = [c_1, c_2, c_3, c_4] \]
\[ c_2p(c, 4) \rightarrow \]
\[ [c_1, 1/2(c_1^2-2c_2), 1/6(c_1^3-3c_1c_2+3c_3), \]
\[ 1/24(c_1^4-4c_1^2c_2+4c_1c_3+2c_2^2-4c_4) \]
\[ :: \text{Vector}(\mathbb{Q}[c_1, c_2, c_3, c_4]) \]
\[ p = [x_1, x_2, x_3, x_4] \]
\[ p_2c(p, 4) \rightarrow \]
\[ [x_1, 1/2(x_1^2-2x_2), 1/6(x_1^3-6x_1x_2+12x_3), \]
\[ 1/24(x_1^4-12x_1^2x_2+48x_1x_3+12x_2^2-144x_4) \]
\[ :: \text{Vector}(\mathbb{Q}[x_1, x_2, x_3, x_4]) \]

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 \( x^n - c_1x^{n-1} + \ldots + (-1)^n c_n \), 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:

\begin{verbatim}
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 = c_2p(c, r)
[k!*p.k with k in range(p)]
\end{verbatim}

\[ f = x^4 + a*x^2 + b*x + c \]
\[ newton_sums(f, x, 10) \rightarrow \]
\[ [0, -2a, -3b, 2a^2-4c, 5ab, -2a^3+6ac+3b^2, \]
\[ -7a^2b+7bc, 2a^4-8a^2c-8ab^2+4c^2, \]
\[ 9a^3b-18abc-3b^3, \]
\[ -2a^5+10a^3c+15a^2b^2-2ac^2-10b^2c \]
\[ :: \text{Vector}(\mathbb{Z}[a, b, c]) \]

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 \textsc{omega} can evaluate very abstract logic and mathematical expressions, which are syntactically close to familiar formulas, like

\[ P = \{ (a, b) \text{ suchthat } f(a, b) = 0 \text{ with } (a, b) \in \mathbb{F}^2 \} \]

On the other hand, all the objects in \( \Omega \) (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 \( \Omega \) 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 \textsc{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 \textsc{omega}. There are two main aspects of this facility. One is related to commands of the form

\begin{verbatim}
babel «latin»
\end{verbatim}

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

\begin{verbatim}
# Latin.dic file

cum -> with

talisut -> suchthat

primus? -> prime?
\end{verbatim}

we could write a file like

\begin{verbatim}
# Programming in latin

babel «latin»

n=10000

p=(k talisut primus?(k) cum k in 1..n)
\end{verbatim}

which would be evaluated as
# Programming in English

\begin{verbatim}
n = 10000
p = \{k \text{ such that } \text{prime}(k) \text{ with } k \text{ in } 1..n\}
\end{verbatim}

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 `label «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

\textit{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 \textit{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 projecte O MEGA: el mòdul d’enters i manipulador simbòlic. FME/UPC, September 1998.

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 identify 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