Efficient programming in maple: a case study

Bruno Salvy

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1)39 63 55 11

Rapports de Recherche

N°1611

Programme 2

*Calcul symbolique, Programmation
et Génie logiciel*

EFFICIENT PROGRAMMING IN MAPLE: A CASE STUDY

Bruno SALVY

Février 1992

Efficient Programming in Maple: A Case Study

Bruno Salvy

Abstract

Studying the computation of orbits of some given group acting on a set, we show how successive refinements of a simple program provide a speedup factor of up to 200. The initial program cannot solve the problem at all; it is first converted into a program which needs one c.p.u. week and after all the optimizations have been performed, the computation takes only four hours. Although the problem is specific, we shall show that many of the optimizations we use are of interest for other problems as well.

Programmation Efficace en Maple : Un Exemple

Résumé

Sur un calcul d'orbites d'un groupe donné agissant sur un ensemble, nous décrivons des améliorations successives d'un programme simple qui mènent à une accélération par un facteur atteignant 200. Le programme de départ est incapable de résoudre le problème, il est d'abord transformé en un programme qui effectue l'ensemble du calcul en une semaine c.p.u. Une fois toutes les optimisations effectuées, le calcul ne nécessite plus que quatre heures. Bien qu'il s'agisse là d'un problème particulier, nous montrons qu'un bon nombre des optimisations utilisées sont d'un intérêt plus général.

Efficient Programming in Maple: A Case Study

Bruno Salvy

Algorithms Project
INRIA Rocquencourt
78153 Le Chesnay
France

Abstract

Studying the computation of orbits of some given group acting on a set, we show how successive refinements of a simple program provide a speedup factor of up to 200. The initial program cannot solve the problem at all, it is first converted into a program which needs one c.p.u. week and after all the optimizations have been performed, the computation takes only four hours. Although the problem is specific, we shall show that many of the optimizations we use are of interest for other problems as well.

1 The mathematical problem

This problem, posed to us by B. Courteau, arises in algebraic coding theory (see [1, 4] for more details). Two matrices are given: an $n_G \times n$ matrix G of integers in $\{1, 2, \ldots, n\}$ and an $m \times n$ matrix K of elements of $\{0, 1\}$; m is less than n. The rows of G represent permutations of $\{1, 2, \ldots, n\}$. In simple words, the computation we need to perform goes as follows:

1. each integer in $\{0, \ldots, 2^m - 1\}$ is converted into a vector in $(\mathbb{Z}/2\mathbb{Z})^n$ by considering its binary representation and adding zeroes at the beginning;
2. to such vector are associated n_G vectors of size n by permuting the coordinates according to the rows of G;
3. each of the vectors is multiplied by the matrix K, yielding vectors of size m;
4. the coordinates of these new vectors are reduced modulo 2, and this last vector is translated back into an integer in $\{0, \ldots, 2^m - 1\}$.

In this way, with each integer in $\{0, \ldots, 2^m - 1\}$ there are associated n_G not necessarily distinct integers, to which the same process can be applied. Since this set is finite, after a finite number of steps no new elements are produced and $\{0, \ldots, 2^m - 1\}$ is thus split into several subsets, called *orbits*, and these are the sets we want to compute. To make the idea more precise, in the

specific case we are interested in, K and G are the following 16×32 and 45×32 matrices:

$$K = \begin{bmatrix}
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1&1&1&1&1&1&0\\
0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&0&0&1&1&0&0&1&0&0&0&0&0&1\\
0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&1&0&1&0&1&0&0&1&0&0&0&0&1\\
0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&0&1&1&0&0&1&0&0&1&0&0&0&1\\
0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&1&0&1&1&0&0&1&1&0&0&0&0&1&0&0&1\\
0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&1&0&1&0&1&0&1&0&1&0&0&0&0&1&0&1\\
0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&1&0&0&1&1&0&0&1&1&0&0&0&0&0&1&1\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&1&1&1&0&0&0&0&0&0&1&1&0&1&0&0&1\\
0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&1&1&0&1&0&0&0&0&0&1&0&1&0&1&0&1\\
0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&1&0&1&1&0&0&0&0&0&0&1&1&0&0&1&1\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&1&1&1&0&0&0&0&0&0&0&0&1&1&1&1\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&1&1&1&0&1&1&0&1&0&0&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&1&1&0&1&1&0&1&0&1&0&1&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&1&0&1&1&0&1&1&0&0&1&1&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&1&1&1&0&0&0&1&1&1&1&1
\end{bmatrix},$$

$$G = \begin{bmatrix}
1&2&4&3&5&6&17&8&9&18&11&12&13&22&15&16&7&10&19&21&20&14&23&25&24&26&28&27&29&30&32&31\\
1&2&3&4&6&5&17&8&9&10&19&12&13&14&23&16&7&18&11&21&20&22&15&25&24&26&27&28&30&29&32&31\\
1&2&3&4&7&17&5&8&9&10&20&12&13&14&24&16&6&18&21&11&19&22&25&15&23&26&27&28&31&32&29&30\\
1&2&3&4&5&6&7&9&8&18&19&12&13&14&15&26&17&10&11&21&20&22&23&24&25&16&28&27&30&29&32&31\\
1&2&3&4&5&6&7&10&18&8&20&12&13&14&15&27&17&9&21&11&19&22&23&24&25&28&16&26&31&32&29&30\\
1&2&3&4&5&6&7&11&19&20&8&12&13&14&15&29&17&21&9&10&18&22&23&24&25&30&31&32&16&26&27&28\\
1&2&3&4&5&6&7&8&9&10&11&13&12&22&23&26&17&18&19&20&21&14&15&25&24&16&28&27&30&29&32&31\\
1&2&3&4&5&6&7&8&9&10&11&14&22&12&24&27&17&18&19&20&21&13&25&15&23&28&16&26&31&32&29&30\\
1&2&3&4&5&6&7&8&9&10&11&15&23&24&12&29&17&18&19&20&21&25&13&14&22&30&31&32&16&26&27&28\\
1&2&3&4&5&6&7&8&9&10&11&16&26&27&29&12&17&18&19&20&21&28&30&31&32&13&14&22&15&23&24&25\\
1&3&2&4&5&7&6&8&10&9&11&12&14&13&15&16&17&18&20&19&21&22&24&23&25&27&26&28&29&31&30&32\\
1&5&3&7&2&6&4&8&11&10&9&12&15&14&13&16&17&20&19&18&21&24&23&22&25&29&27&31&26&30&28&32\\
1&8&3&10&5&11&7&2&9&4&6&12&16&14&15&13&20&18&19&17&21&27&29&24&31&26&22&28&23&30&25&32\\
1&12&3&14&5&15&7&8&16&10&11&2&13&4&6&9&24&27&29&20&31&22&23&17&25&26&18&28&19&30&21&32\\
2&1&4&3&6&5&17&9&8&18&19&13&12&22&23&26&7&10&11&21&20&14&15&25&24&16&28&27&30&29&32&31\\
3&4&1&2&7&17&5&10&18&8&20&14&22&12&24&27&6&9&21&11&19&13&25&15&23&28&16&26&31&32&29&30\\
4&3&2&1&17&7&6&18&10&9&21&22&14&13&25&28&5&8&20&19&11&12&24&23&15&27&26&16&32&31&30&29\\
5&6&7&17&1&2&3&11&19&20&8&15&23&24&12&29&4&21&9&10&18&25&13&14&22&30&31&32&16&26&27&28\\
6&5&17&7&2&1&4&19&11&21&9&23&15&25&13&30&3&20&8&18&10&24&12&22&14&29&32&31&26&16&28&27\\
7&17&5&6&3&4&1&20&21&11&10&24&25&15&14&31&2&19&18&8&9&23&22&12&13&32&29&30&27&28&16&26\\
17&7&6&5&4&3&2&21&20&19&18&25&24&23&22&32&1&11&10&9&8&15&14&13&12&31&30&29&28&27&26&16\\
8&9&10&18&11&19&20&1&2&3&5&16&26&27&29&12&21&4&6&7&17&28&30&31&32&13&14&22&15&23&24&25\\
9&8&18&10&19&11&21&2&1&4&6&26&16&28&30&13&20&3&5&17&7&27&29&32&31&12&22&14&23&15&25&24\\
10&18&8&9&20&21&11&3&4&1&7&27&28&16&31&14&19&2&17&5&6&26&32&29&30&22&12&13&24&25&15&23\\
18&10&9&8&21&20&19&4&3&2&17&28&27&26&32&22&11&1&7&6&5&16&31&30&29&14&13&12&25&24&23&15\\
11&19&20&21&8&9&10&5&6&7&1&29&30&31&16&15&18&17&2&3&4&32&26&27&28&23&24&25&12&13&14&22\\
19&11&21&20&9&8&18&6&5&17&2&30&29&32&26&23&10&7&1&4&3&31&16&28&27&15&25&24&13&12&22&14\\
20&21&11&19&10&18&8&7&17&5&3&31&32&29&27&24&9&6&4&1&2&30&28&16&26&25&15&23&14&22&12&13\\
21&20&19&11&18&10&9&17&7&6&4&32&31&30&28&25&8&5&3&2&1&29&27&26&16&24&23&15&22&14&13&12\\
12&13&14&22&15&23&24&16&26&27&29&1&2&3&5&8&25&28&30&31&32&4&6&7&17&9&10&18&11&19&20&21\\
13&12&22&14&23&15&25&26&16&28&30&2&1&4&6&9&24&27&29&32&31&3&5&17&7&8&18&10&19&11&21&20\\
14&22&12&13&24&25&15&27&28&16&31&3&4&1&7&10&23&26&32&29&30&2&17&5&6&18&8&9&20&21&11&19\\
22&14&13&12&25&24&23&28&27&26&32&4&3&2&17&18&15&16&31&30&29&1&7&6&5&10&9&8&21&20&19&11\\
15&23&24&25&12&13&14&29&30&31&16&5&6&7&1&11&22&32&26&27&28&17&2&3&4&19&20&21&8&9&10&18\\
23&15&25&24&13&12&22&30&29&32&26&6&5&17&2&19&14&31&16&28&27&7&1&4&3&11&21&20&9&8&18&10\\
24&25&15&23&14&22&12&31&32&29&27&7&17&5&3&20&13&30&28&16&26&6&4&1&2&21&11&19&10&18&8&9\\
25&24&23&15&22&14&13&32&31&30&28&17&7&6&4&21&12&29&27&26&16&5&3&2&1&20&19&11&18&10&9&8\\
16&26&27&28&29&30&31&12&13&14&15&8&9&10&11&1&32&22&23&24&25&18&19&20&21&2&3&4&5&6&7&17\\
26&16&28&27&30&29&32&13&12&22&23&9&8&18&19&2&31&14&15&25&24&10&11&21&20&1&4&3&6&5&17&7\\
27&28&16&26&31&32&29&14&22&12&24&10&18&8&20&3&30&13&25&15&23&9&21&11&19&4&1&2&7&17&5&6\\
28&27&26&16&32&31&30&22&14&13&25&18&10&9&21&4&29&12&24&23&15&8&20&19&11&3&2&1&17&7&6&5\\
29&30&31&32&16&26&27&15&23&24&12&11&19&20&8&5&28&25&13&14&22&21&9&10&18&6&7&17&1&2&3&4\\
30&29&32&31&26&16&28&23&15&25&13&19&11&21&9&6&27&24&12&22&14&20&8&18&10&5&17&7&2&1&4&3\\
31&32&29&30&27&28&16&24&25&15&14&20&21&11&10&7&26&23&22&12&13&19&18&8&9&17&5&6&3&4&1&2\\
32&31&30&29&28&27&26&25&24&23&22&21&20&19&18&17&16&15&14&13&12&11&10&9&8&7&6&5&4&3&2&1
\end{bmatrix}.$$

Here K is a parity check matrix for a code equivalent to the second order Reed-Muller code of length 32 and the rows of G are permutations corresponding to a set of generators of an automorphism group of this code which is isomorphic to the general affine group GA(5) (see [4, p.399]). The mathematical problem is to study the action of this set of generators on the space of syndromes of the code, but its understanding is not necessary for our purpose.

2 Naive programming

In this section we write the (short) program such as one would expect a typical top-level user of Maple to write it. This is almost the form in which the program was brought to our attention for optimization.

The matrices K and G are stored in a file, we read them in and put their dimensions in global variables, using procedures from the linear algebra (`linalg`) package:

```
read `data.m`;
m:=linalg[rowdim](K);
n:=linalg[coldim](K);
ng:=linalg[rowdim](G);
```

The following procedure is used to translate integers into vectors:

```
vec:=proc(x) local v,i;
  v:=convert(x+2^n,base,2);
  [v[i]$i=1..n]
end;
```

What happens here is that Maple already knows how to convert an integer p to its binary representation, but returns a list of length $\log_2(p)$, whereas we need an output of fixed length n, hence the cheat by adding 2^n to the integer and taking into account only the first n binary digits.

Any reader who is familiar with Maple will notice several easy improvements here, which we shall return to in the next section.
In the same vein, one can write the reciprocal operation:

```
num:=proc(v) local i; sum(v[i]*2^(i-1),i=1..m) end;
```

The following procedure computes the action of a vector f of G on a integer p: first p is converted into a vector, then its coordinates are permuted by the vector f under consideration and finally the product by K is computed modulo 2.

```
action:=proc(f,p)
local x,y,t,i,j,k;
  x:=vec(p);
  y:=[x[f[k]]$k=1..n];
  t:=array(1..m,sparse);
  for i to m do for j to n do t[i]:=t[i]+K[i,j]*y[j] mod 2 od od;
  num(t)
end:
```

Now, we shall need a way to call `action` on each line of G. This will be done by the following:

```
gg:=proc(z,i) action(linalg[subvector](G,i,1..n),z) end:
```

Finally, we compute the orbits by iterating this procedure until no newer elements are found:

```
orbit:=proc(z)
local orb,prevorb,newelts;
    prevorb:={};
    orb:={z};
    while orb <> prevorb do
      newelts:=orb minus prevorb;
      prevorb:=orb;
      orb:=`union`(`map(gg,newelts,i)`$`i`=1..ng) union orb
    od;
    orb
end;
```

Here we have used the $n-$ary union of sets (`` `union`(a,b,..) ``) as well as the more customary binary infix operator (`a union b`).

The mathematical problem has thus been translated in a very straightforward way into 30 lines of Maple code. The timings of this version of the program are indicated in the first line

of Table 1 which is given in the appendix. In this table, the head of the columns indicate the number of elements of each orbit, and the entries in the table are the computing time in seconds of each version of the program for each orbit. These timings were obtained on a Sun4/75 (SparcStation 2) with 64 Mo of memory. Concerning this particular version of the program, one can see that the total amount of computing time necessary for the whole decomposition is huge (more than 24 c.p.u. hours for the orbit of size 4960). Besides the largest orbits cannot be obtained by this version which uses too much memory. One then understands the need for a serious optimization.

3 Basic improvements

The program given in our previous section would not have been written by a user accustomed to programming in Maple. We shall review in this section a few programming habits that make Maple code more efficient.

3.1 The `seq` procedure

In versions of Maple prior to MapleV, the `$` construct was the only way to produce a sequence of expressions, with a syntax which is exemplified in the procedures of our previous section (see [2] for more details). However, whereas Maple expressions are not fully evaluated in procedures (this was introduced in Maple4.0 and led to a general speedup of the system), the `$` construct is an exception to this rule, hence a loss of efficiency, and the need for an unpleasant quoting of its arguments as done in our procedure `orbit`.

In MapleV, a new construct named `seq` has been introduced which does not suffer `$`'s flaws. The instruction `seq(f(i),i=m..n)` is equivalent to

t:=NULL;
for i **from** m **to** n **do** t:=t,f(i) **od**;

but it is more efficient than the above because it needs only one call to Maple's evaluator, a fact which is always interesting in an interpreted language.

3.2 Bad and good uses of `sum`

Maple's `sum` procedure is a discrete analogue to `int`, which computes definite and indefinite integrals. Similarly, `sum` computes definite and indefinite sums, such as:[1]

```
> sum(binomial(binomial(k,2),2)*binomial(2*n-k,n),k=0..n);
                                  2       3
                      (96 n - 144 n  + 48 n ) (2 n + 1) binomial(2 n, n)
                1/8 ---------------------------------------------------
                                (5 + n) (n + 4) (3 + n) (n + 2)
```

However, perhaps due to its name, `sum` is also often used by beginners to compute sums of values (instead of sequences). In this case, `sum` prepares to compute an indefinite sum until it realizes there is nothing to compute, and a (small) amount of time is wasted here. Instead, the proper way to compute such a sum is to convert the list of expressions from the type `list` to the type `+`.

3.3 The kernel and the libraries

Maple is based on a small kernel written in C, and on large libraries of Maple code which are interpreted. Whenever possible then, it is generally more efficient to have as much of the

[1] This example is taken from the "exam problem" section of chapter 5 of [3], where the algorithm is also described.

computation done by functions in the kernel. The `op` procedure enables a user to decide whether a procedure is defined in the kernel or in the library, the procedures of the kernel being tagged as `builtin`. For instance, `modp` which is the prefix unsigned version of `mod` is defined in the kernel:

```
> op(modp);
proc() options builtin; 100 end
```

As a consequence, and since `modp` can be applied to lists as well as integers, the procedure `action` should be rewritten with fewer calls to `mod`.

3.4 Time versus memory

The initial version of the program is unable to perform the computation for the largest orbits. This means that one should be careful to program in such a way that memory is spared as much as possible.

The `map` and `seq` constructs. Another use of the `seq` construct described above takes a structure as second argument. Thus `seq(f(i),i=u)` is equivalent to

t:=NULL;
for i **to** nops(u) **do** t:=t,f(op(i,u)) **od**;

It would seem that in the case when f has only one argument, this is almost equivalent to `map(f,[op(u)])`. However, it appears that `map` allows a garbage collection to occur between two evaluations of f, whereas `seq` does not. In practice, this means that when memory requirements become important, it is always better to use `map` if possible.

Double quotes and local variables. In Maple, three local variables are provided, named `"`, `""` and `"""`. Their values are the last expression evaluated, the penultimate one and its predecessor. Their evaluation rule is different from ordinary local variables, since they are fully evaluated. Thus there results a loss in efficiency in using them to address structured objects. Besides, their use makes difficult to read programs. However, in special circumstances where one local variable requires a huge amout of memory, they can help in saving a local variable, hence memory, hence speed because of garbage collections. We shall use them this way in `orbit`.

3.5 The new program

Based on these four remarks and on a few simpler ones, we can rewrite a shorter and faster version of our previous program:

```
vec:=proc(x) local v;
  v:=convert(x,base,2);
  [op(v),0$n-nops(v)]
end:

num:=proc(v) local i; convert([seq(v[i]*2^(i-1),i=1..m)],`+`) end:

action:=proc(f,p)
local x,y,i,j,k;
  x:=vec(p);
  y:=[seq(x[f[k]],k=1..n)];
  num(modp([seq(convert([seq(K[i,j]*y[j],j=1..n)],`+`),i=1..m)],2))
end:
```

```
gg:=proc(u,z) action(linalg[subvector](G,u,1..n),z) end:

orbit:=proc(z)
local orb,newelts,i,s;
    s:={$1..ng};
    newelts:={z};orb:=newelts;
    while newelts <> {} do
        newelts:={};
        for i in """ do newelts:=(map(gg,s,i) minus orb) union newelts od;
        orb:=orb union newelts
    od;
    orb
end:
```

The timings of this version are indicated in line "version 2" of Table 1. *We can see that these direct syntactic modifications lead to a speedup by a factor of 2.5*, but do not yet enable a full solution of this problem in a reasonable amount of time.

4 Optimizations related to the problem

Before proceeding with more elaborate Maple tools, we shall have a closer look at the particular problem being studied in order to get an optimization at a more global level.

Firstly it is obvious that there is a lot of converting to and from between vectors and integers while the integer format is really only needed at the user level (i.e., the output of `orbit`). Many operations can be avoided by not doing this conversion. This leads us to modify `action` as follows:

```
action:=proc(f,x)
local y,i,j,k;
  y:=[seq(x[f[k]],k=1..n)];
  modp([seq(convert([seq(K[i,j]*y[j],j=1..n)],'+'),i=1..m),0$(n-m)],2)
end:
```

while changing the beginning of `orbit` with the line

```
    v:=convert(z,base,2);newelts:={[op(v),0$(n-nops(v))]};
```

Secondly it might happen that several of the vectors obtained by applying G to the elements of `newelts` are actually identical, in which case it would be useless to compute several times the product by K. Then one is led to consider the following simpler version of `action`:

```
synd:=proc(y) local i,j;
  [seq(convert([seq(K[i,j]*y[j],j=1..n)],'+'),i=1..m),0$(n-m)]
end:

action:=proc(u) local i,j,x;
  modp(map(synd,{seq(seq([seq(x[G[i,j]],j=1..n)],i=1..ng),x=u)}),2)
end:
```

However, although the version of the program is about 3 times as fast as the previous one on the small orbits, it loses by far on the larger ones. The reason for this is interesting. While the previous program could work in constant space independently of the size of the orbit (except of course the procedure `orbit`), this is not the case anymore with this program which has to wait till it has computed the set of the applications of G to all the elements of u before computing K.

This set can be very large, which leads Maple to allocate a lot of memory space and to perform a lot of garbage collections. Consequently, this optimization cannot be performed.

The timings of this version of the program where just the superfluous conversions between numbers and vectors have been suppressed are indicated in line 3 of Table 1, and show a speedup over version 2 of approximately a factor 1.1 on the small orbits and a loss by a factor 1.4 on the largest ones. Once again, this is due to the larger amount of memory required by the representation of the numbers as vectors. We shall however keep this change which will pay off later.

5 Profiling the computation

Maple provides a tool for profiling procedures. As usual the idea is to concentrate the optimization work on those procedures that are the most expensive. In order to get a better information from this, we decompose `action` into two procedures, incorporate `gg` into `action` and modify `orbit` accordingly:

```
action:=proc(z,i) local k; modp(synd([seq(z[G[i,k]],k=1..n)]),2) end:
```

where synd is the procedure we introduced and gave up in our previous optimization. We then profile all our routines:

```
> readlib(profile)(num,synd,action,orbit):
```

and start a short computation

```
> orbit(1):
```

we get as output of `showprofile()` a table which reads as:

function	depth	calls	time	time%	words
orbit	1	1	185.266	99.9908%	11196595
action	1	1440	180.246	97.2815%	10869533
synd	1	1440	166.565	89.8976%	10143663
num	1	32	0.167	0.0901%	12747
total			185.283		11198546

and this indicates that most of the time and memory is spent in the 3-line long procedure `synd`, and that one should optimize this procedure first, and not lose time on `num` for instance, although it could probably be optimized.

6 Precomputations

The optimization we are going to introduce is one that is often overlooked by programmers in computer algebra who are used to thinking in symbolic terms. However, even though computer algebra systems are good at symbolic computations, they are always faster when the computation is purely numerical. The idea here is to perform some of the operations symbolically *before* the actual computation. For instance, since `synd` appeared as the most expensive procedure of our program, we can rewrite it as:

```
synd:=subs(_BODY=[seq(convert([seq(K[i,j]*y[j],j=1..n)],`+`),i=1..m),0$(n-m)],
  proc(y) _BODY end):
```

What happens is that the evaluation of the $m \cdot n$ products will be performed only once, when `synd` is evaluated, and this does not take any noticeable amount of time. If now we wish to look at the body of the procedure `synd`, it has become a routine with only additions:

```
> lprint(op(synd));
proc (y) [y[1]+y[17]+y[18]+y[19]+y[20]+y[21]+y[22]+y[23]+y[24]+y[25]+y[26]+y[
27]+y[28]+y[29]+y[30]+y[31], y[2]+y[17]+y[18]+y[19]+y[22]+y[23]+y[26]+y[32], y[
3]+y[17]+y[18]+y[20]+y[22]+y[24]+y[27]+y[32], y[4]+y[17]+y[18]+y[21]+y[22]+y[25
]+y[28]+y[32], y[5]+y[17]+y[19]+y[20]+y[23]+y[24]+y[29]+y[32], y[6]+y[17]+y[19]
+y[21]+y[23]+y[25]+y[30]+y[32], y[7]+y[17]+y[20]+y[21]+y[24]+y[25]+y[31]+y[32]
, y[8]+y[18]+y[19]+y[20]+y[26]+y[27]+y[29]+y[32], y[9]+y[18]+y[19]+y[21]+y[26]+
y[28]+y[30]+y[32], y[10]+y[18]+y[20]+y[21]+y[27]+y[28]+y[31]+y[32], y[11]+y[19]
+y[20]+y[21]+y[29]+y[30]+y[31]+y[32], y[12]+y[22]+y[23]+y[24]+y[26]+y[27]+y[29]
+y[32], y[13]+y[22]+y[23]+y[25]+y[26]+y[28]+y[30]+y[32], y[14]+y[22]+y[24]+y[25
]+y[27]+y[28]+y[31]+y[32], y[15]+y[23]+y[24]+y[25]+y[29]+y[30]+y[31]+y[32], y[
16]+y[26]+y[27]+y[28]+y[29]+y[30]+y[31]+y[32], 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0] end
```

In order to apply a similar trick to `action`, we need `action` to perform its computation for all the lines in G, thus the new code for `action` and `orbit` is:

```
action:=subs(_SUBEXPR={seq([seq(z[G[i,j]],j=1..n)],i=1..ng)},
    proc(z) modp(map(synd,_SUBEXPR),2) end):

orbit:=proc(z)
local orb,v,newelts,i;
    v:=convert(z,base,2);newelts:={[op(v),0$(n-nops(v))]};
    orb:=newelts;
    while newelts <> {} do
        newelts:={};
        for i in """ do newelts:=(action(i) minus orb) union newelts od;
        orb:=orb union newelts
    od;
    map(num,orb)
end:
```

and the expanded code for `action` is now 124 line long. While we are at it, we can do the same optimization on `num` as well:

```
num:=subs(_BODY=convert([seq(v[i]*2^(i-1),i=1..m)],`+`),proc(v) _BODY end):
```

The timings of this version are given in line 4 of Table 1, indicating a speedup ranging from a factor 1.47 to 2.85. A profile of the same computation as before now gives:

function	depth	calls	time	time%	words
orbit	1	1	12.200	99.7302%	799513
action	1	32	12.017	98.2343%	781341
synd	1	1135	7.186	58.7427%	442528
num	1	32	0.034	0.2779%	3301
total			12.233		801470

A comparison with our previous profile indicates a fantastic saving of memory consumption (by a factor 14) from which most of the speedup results.

Another optimization can be performed at this level which was not possible previously: we can get rid of the trailing zeroes in `synd` and work only with vectors of size m. This is achieved by noticing that the coordinates of indices between $m+1$ and n are always 0 during the computation. Performing the simplification when `action` is evaluated leads to the new code:

```
action:=subs(
    _SUBEXPR=subs([seq(z[i]=0,i=m+1..n)],{seq([seq(z[G[i,j]],j=1..n)],i=1..ng)}),
```

proc(z) modp(map(synd,_SUBEXPR),2) **end**):

together with trivial modifications in `synd` and `orbit`.

This gives us the timings of line 5 of Table 1, which show that by doing this, we slowed down the computation by a factor of approximately 1.2. However, we shall keep this change which will enable better speedups in our next optimization step, which is the precomputation of the composition of G and K.

From our previous program, it is a simple matter to write the procedure computing this composition:

action:=subs(_BODY=action(u), **proc**(u) modp(_BODY,2) **end**):

The resulting timings are indicated in line 6 of Table 1. As it turns out, this one line modification is by far our best optimization, giving a speedup by a factor ranging from 4 to 9 over version 4 of the program.

7 The `optimize` routine

Our last step consists in observing that since `action` produces n_G vectors with m coordinates, there is a good chance that several of these coordinates are identical and could be computed only once. In order to do so, we want to use Maple's `optimize` routine. This routine is called with an expression or an array or a list of equations, and yields an optimized sequence of equalities for the computation of its argument. The principal optimizations that are performed are the sharing of common subexpressions and binary powering. Also, a companion routine `optimize/makeproc` takes a computation sequence produced by `optimize` and yields a procedure performing the evaluation of the optimized expression.

In our particular problem, the use of `optimize` is not completely straightforward because the expression we wish to optimize—`action(x)`—is a set of lists, and `optimize` accepts neither of these. Therefore we have to substitute dummy functions for them, calculate the computation sequence, substitute back lists and sets in place of our dummy functions, insert our mod 2 there and call `optimize/makeproc` on this. Thus the first steps are as follows:

u:=subs(U=**proc**()'modp({args},2)'**end**,V=**proc**()[args]**end**,
[readlib(optimize)(U(seq(V(op(j)),j=action(z))))]):

Now when we try to call `optimize/makeproc` on u, it realizes that we have done some non standard tampering and complains that u "must be a computation sequence". Since we know what we are doing is valid, we have to bypass some type-checking in `optimize/makeproc`. In order to do that, we look at its code (this is possible in Maple):

```
> interface(verboseproc=2):op(`optimize/makeproc`);
```

and realize that all we have to do is:

readlib(`optimize/makeproc`):
action:=`optimize/make`(u,[z],map(`optimize/locals`,u)):

it is interesting to have a look at the 150 line long procedure `action` which is thus generated:

```
> op(action);
proc(z)
local t3,t4,t16,t17,t18,t20,t27,t33,t35,t37,t38,t39,t43,t47,t54,t59,t61,t63,
    t65,t66,t69,t74,t81,t88,t95,t98,t99,t100,t103,t104,t105,t108,t109,t110,
    t113,t114,t117,t122,t123,t124,t125,t128,t129,t132,t137,t146,t147,t148,t149,
    t150,t153,t154,t157,t162,t171,t189;
    t3 := z[4]+z[7]+z[10]+z[14];
    t4 := z[3]+z[7]+z[10]+z[14];
    t16 := z[6]+z[7]+z[11]+z[15];
```

```
    ...
    52 similar lines go there
    ...

  t189 := modp({[z[5]+z[4]+z[9]+z[10]+z[13]+z[14]+z[16],t98,t3,z[4],t108,
      t109,t110,t37,z[9],z[10],t54,t65,z[13],z[14],t81,z[16]],[
      z[9]+z[3]+z[5]+z[7]+z[12]+z[14]+z[15],t103,z[3],t4,z[5],t17,z[7],t128,

    ...
    another 83 similar lines
    ...

      z[8],z[9],z[10],t54,t153,t147,t148,t162,t150]},2)
end
```

This new version is tested in line 7 of Table 1. The speedup over version 6 ranges from a factor 3.5 to a factor 7.7.

Conclusion

Even though each particular problem allows specific optimizations, there are general tools that can be applied with benefit to a large class of programs. Unfortunately, many of these tools are not widely known since there does not exist any documentation of this kind.

Another interesting feature of this example is that none of the optimizations we performed gives much speedup, while the cumulated speedup is impressive, meaning that each of our modifications has to be part of the "bag of tricks" of any Maple programmer.

Finally, it is necessary to emphasize that all of this is only a programmer's work, not considering what could exist as better algorithms; and another important point is that computer algebra systems are not necessarily the most suitable tools for this particular kind of computations, but it might nonetheless happen as a subpart in a more purely symbolic problem, necessitating the use of a computer algebra system.

References

[1] Camion, P., Courteau, B., and Montpetit, A. Weight distributions of cosets of 2-error correcting binary BCH codes of length 15, 63 and 255. *IEEE Transactions on Information Theory* (1992). To appear.

[2] Char, B. W., Geddes, K. O., Gonnet, G. H., Leong, B. L., Monagan, M. B., and Watt, S. M. *MAPLE V: Language Reference Manual.* Springer-Verlag, New York, 1991.

[3] Graham, R., Knuth, D., and Patashnik, O. *Concrete Mathematics.* Addison Wesley, 1989.

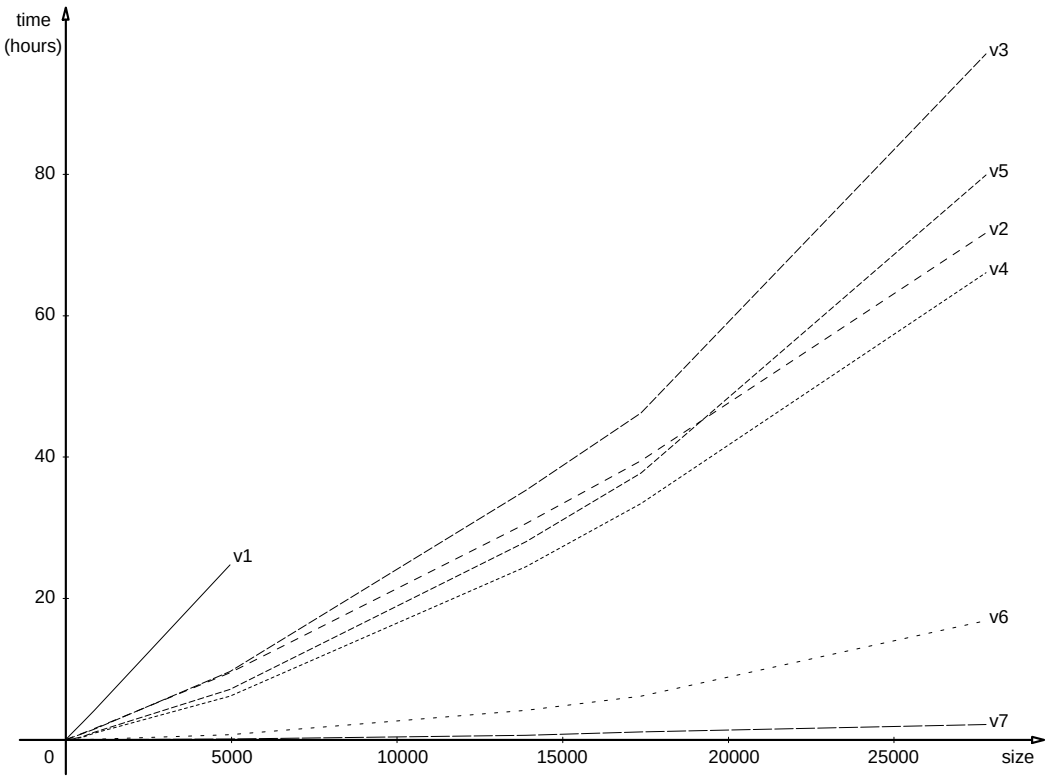
[4] MacWilliams, F. I., and Sloane, N. J. A. *The Theory of Error-Correcting Codes.* North-Holland, 1977.

APPENDIX

A Timings of the programs

	1	32	155	496	868	4960	13888	17360	27776
version 1	17	546	2649	8602	15055	89087	n.a.	n.a.	n.a.
version 2	6	212	1035	3304	5779	34175	110137	142092	258199
version 3	6	194	944	3064	5423	34861	127029	166554	349309
version 4	0.1	68	278	1286	3373	22198	88211	120401	237963
version 5	0.1	84	308	1491	3916	25758	100960	135844	287662
version 6	0.1	7	46	159	221	2598	15031	22283	60768
version 7	0.1	2	8	28	53	498	2383	4088	7853

Table 1: Timings in seconds of the programs on each orbit according to its size



The curves corresponding to Table 1

B The final program

```
read `data.m`:
m:=linalg[rowdim](K):
n:=linalg[coldim](K):
ng:=linalg[rowdim](G):

num:=subs(_BODY=convert([seq(v[i]*2^(i-1),i=1..m)],`+`),proc(v) _BODY end):

synd:=proc(y) local i,j;
    [seq(convert([seq(K[i,j]*y[j],j=1..n)],`+`),i=1..m)] end:

action:=proc(z) local i,j;
    map(synd,subs([seq(z[i]=0,i=m+1..n)],[seq([seq(z[G[i,j]],j=1..n)],i=1..ng)]))
end:

u:=subs(U=proc()'modp({args},2)'end,V=proc()[args]end,
    [readlib(optimize)(U(seq(V(op(j)),j=action(z))))]):
readlib(`optimize/makeproc`):
action:=`optimize/make`(u,[z],map(`optimize/locals`,u)):

orbit:=proc(z)
local orb,v,newelts,i;
    v:=convert(z,base,2);
    newelts:={[op(v),0$(m-nops(v))]};orb:=newelts;
    while newelts <> {} do
        newelts:={};
        for i in """ do newelts:=(action(i) minus orb) union newelts od;
        orb:=orb union newelts
    od;
    map(num,orb)
end:
```