ELSEVIER Journal of Pure and Applied Algebra 119 (1997) 237–253

JOURNAL OF PURE AND APPLIED ALGEBRA

Computation of Hilbert–Poincaré series

Anna M. Bigatti *

Dipartimento di Matematica, Università di Genova, Genova, Italy

Communicated by L. Robbiano; received 22 March 1995

Abstract

We describe a new algorithm for computing standard and multi-graded Hilbert–Poincaré series of a monomial ideal. We compare it with different strategies along with implementation details and timing data. © 1997 Elsevier Science B.V.

1991 Math. Subj. Class.: Primary 13D40, 13-04, 13P99; Secondary 13P10, 68Q40

1. Introduction

Many papers have been written to present different algorithms to compute the Hilbert–Poincaré series of a monomial ideal. In this paper we want to do something which previous papers on this topic have not completely done: we give a very detailed description of our work along with implementation details and timing data, explaining how different strategies influence the behaviour of the program, and why sometimes theoretically optimal strategies are not good in practice.

The timings we obtain are almost negligible compared with the corresponding Gröbner basis computation. It follows that the Hilbert–Poincaré series may be fruitfully used as a tool for other computations, for instance in the Hilbert-driven Buchberger algorithm (see [6, 9]).

In Section 2, after setting out the notation, we give the theoretical information defining the main cases we will meet during the computation: pivot-case, splitting-case and base-case. Section 3 briefly describes the structure of the algorithm and in Section 4 we mention the examples on which we will test the different strategies.

Section 5 deals with the pivot-case. The choice of the pivot is the main distinction between the known algorithms. We suggest a new choice and compare it with the

* E-mail: bigatti@dima.unige.it.

others. Then we deal with the interreduction of the power-product lists. Its complexity is quadratic in the length of the list, whereas all the other operations have linear complexity, so it is the most time costly operation in the computation of Hilbert–Poincaré series.

In Section 6 we compare the cost of looking for general splitting-cases with the benefits they might bring, and we describe their specialisation into total-splitting-cases. Section 7 analyses the data structures. We point out the operations we need on polynomials and describe our extended representation for power-products, consisting of dense representation + support + bit-support. Finally, in Section 8, a table is given with the best timings published in [2] and the timings we get with our implementation in the new system CoCoA 3.

2. Definitions, terminology and notation

Let k be a field and $R := k[X_1,\dots,X_N]$ be a $\mathbf{Z}^r$-graded ring where, for some term-ordering σ on $\mathbf{Z}^r$, $\deg X_i >_\sigma 0$ for each $i = 1,\dots,N$ and let $M = \bigoplus_{d\in\mathbf{Z}^r} M_d$ be a finitely generated $\mathbf{Z}^r$-graded module over R. The M_d's are finite dimensional k-vector spaces (see [1, Chapters 10, 11]). Note that whereas traditionally the degree d is a non-negative integer, we allow $d \in \mathbf{Z}^r$, i.e., what we are going to see holds in general for "multigraded" rings and modules and it trivially specializes into the traditional $\mathbf{N}$-graded case with the usual definition of degree.

The function

$$H_M : \mathbf{Z}^r \to \mathbf{N} \quad \text{with } H_M(d_1,\dots,d_r) := \dim(M_{d_1,\dots,d_r})$$

is called the *Hilbert function* of M. The *Hilbert–Poincaré series* of M is

$$HP_M(t) = \sum_{d\in\mathbf{Z}^r} H_M(d)t^d$$

(where $t^d := t_1^{d_1}\cdot\ldots\cdot t_r^{d_r}$) and from the Hilbert–Serre theorem we know that it can be written as

$$HP_M(t) = \frac{Q(t)}{\prod_{i=1,\dots,N}(1 - t^{\deg X_i})}, \tag{1}$$

where $Q(t) \in \mathbf{Z}[t_1,\dots,t_r,t_1^{-1},\dots,t_r^{-1}]$.

The computation of a Hilbert–Poincaré series of an R-module may be reduced, via any Gröbner basis, to the computation of the Hilbert–Poincaré series of some k-algebras of type R/I where I is a monomial ideal (see [2, 3]). In particular it is well known that if I is a (multi)homogeneous ideal in R, then $HP_{R/I} = HP_{R/\mathrm{In}(I)}$ (see [13, 15]) and $\mathrm{In}(I)$ can be computed since it is generated by the leading terms of the elements of any Gröbner basis of I.

The polynomial $Q(t)$ will be denoted by $\langle I\rangle$ (following the notation in [2, 3]) and we will show how to compute it. (For more details about multivariate Hilbert–Poincaré series see [6].)

Example 1. Let $R/(X_1,\dots,X_N)=k$ and

$$\frac{\langle X_1,\dots,X_N\rangle}{\prod_{i=1,\dots,N}(1-t^{\deg X_i})}=HP_k(t)=1.$$

Then

$$\langle X_1,\dots,X_N\rangle=\prod_{i=1,\dots,N}(1-t^{\deg X_i}). \tag{2}$$

In particular if R/I is a standard algebra, i.e., **N**-graded and $\deg X_i=1$ for all $i=1,\dots,N$, then

$$HP_{R/I}(t)=\frac{\langle I\rangle}{(1-t)^N},$$

where $\langle I\rangle\in\mathbf{Z}[t]$. In this case for all sufficiently large d, $H_{R/I}(d)$ is a polynomial in d (see [1, 16]).

Example 2. Let $R=k[X_1,\dots,X_N]$ be standard. Then $R=\bigoplus_{d\in\mathbf{N}}R_d$ where each R_d is minimally generated as a k-vector by all the $\binom{N+d-1}{d}$ power-products of degree d. Therefore,

$$HP_R(t)=\sum_{i=0}^{\infty}\dim(R_d)t^d=\sum_{i=0}^{\infty}\binom{N+d-1}{d}t^d=1/(1-t)^N.$$

Hence $\langle 0\rangle=1$.

After this simple example let us see a few interesting cases which will provide us with a recursive method to compute the Hilbert–Poincaré series of a (multi)graded algebra.

2.1. *Pivot-case*

All the recent algorithms [2–4, 10, 12] are based on the following short exact sequence of graded R-modules (let $\mathscr{P}$ be a monomial of degree $d=(d_1,\dots,d_r)$ which will be called the *pivot* as in [4])

$$0\to R/(I:\mathscr{P})\xrightarrow{\cdot\mathscr{P}}R/I\to R/(I,\mathscr{P})\to 0$$

which yields $HP_{R/I}(t)=HP_{R/(I,\mathscr{P})}(t)+t^d(HP_{R/(I:\mathscr{P})}(t))$ and then

$$\langle I\rangle=\langle I,\mathscr{P}\rangle+t^d\langle I:\mathscr{P}\rangle. \tag{3}$$

If we choose $\mathscr{P}$ strictly dividing at least one of the generators of I, then the total degrees of $(I, \mathscr{P})$ and $(I : \mathscr{P})$ are lower than the total degree of I (where with total degree we mean the sum of the traditional $\mathbf{N}$-degrees in the minimal set of generators).

In [2], $\mathscr{P}$ is chosen to be one of the generators and (3) is read as $\langle I, \mathscr{P}\rangle = \langle I\rangle - t^d\langle I : \mathscr{P}\rangle$. In this way the number of the generators decreases.

So the *pivot-case* reduces the computation of $\langle I\rangle$ to two "simpler" computations (see Section 5). This process is called *horizontal splitting* in [4].

Example 3. Observe that if $\mathscr{P}$ is a non-zero-divisor in R/I, then $I = (I : \mathscr{P})$. Thus, by (3), $\langle I, \mathscr{P}\rangle = (1 - t^{\deg \mathscr{P}})\langle I\rangle$.

Since $X_1, \ldots, X_N$ form a regular sequence it follows that

$$\langle X_1, \ldots, X_N\rangle = (1 - t^{\deg X_1})\langle X_2, \ldots, X_N\rangle = \cdots = \prod_{i=1,\ldots,N} (1 - t^{\deg X_i})\langle 0\rangle.$$

From (2) we have $\langle X_1, \ldots, X_N\rangle = \prod_{i=1,\ldots,N}(1 - t^{\deg X_i})$, therefore

$$\langle 0\rangle = 1 \tag{4}$$

as we have already seen in Example 2 for standard algebras.

2.2. *Splitting-case*

Sometimes it happens that we can partition the set of the indeterminates into disjoint non-empty subsets $\mathbf{X}_1, \ldots, \mathbf{X}_s$ such that each generator of our monomial ideal I belongs to a $k[\mathbf{X}_j]$ for some j. We shall call this a *splitting-case.*

Let $I_j := I \cap k[\mathbf{X}_j]$ and observe that $R/I = k[\mathbf{X}_1]/I_1 \otimes \cdots \otimes k[\mathbf{X}_s]/I_s$.

Since Hilbert–Poincaré series multiply with respect to tensor products it follows that $HP_{R/I}(t) = HP_{k[\mathbf{X}_1]/I_1}(t) \cdots HP_{k[\mathbf{X}_s]/I_s}(t)$.

Then

$$\frac{\langle I\rangle}{\prod_{i=1,\ldots,N}(1 - t^{\deg X_i})} = \frac{\langle I_1\rangle}{\prod_{X_i \in \mathbf{X}_1}(1 - t^{\deg X_i})} \cdots \frac{\langle I_s\rangle}{\prod_{X_i \in \mathbf{X}_s}(1 - t^{\deg X_i})}$$

and therefore

$$\langle I\rangle = \langle I_1\rangle \cdot \cdots \cdot \langle I_r\rangle \tag{5}$$

which is called *vertical splitting* in [4] and *variant* (B) in [2].

2.3. *Base-cases*

Using (3) and (5) we can reduce our computation to simpler ones; now we must choose our base-cases. $\langle 0\rangle = 1$ is a base-case, but it is obviously better if we use a more general base-case in order to avoid some recursion calls.

For example we shall see that we can easily compute $\langle I\rangle$ when I is generated by *simple-powers*, i.e., power-products where only one indeterminate has non-zero power. Then, apart from the actual computation of the base-cases, we shall almost ignore the presence of the simple-powers and, given a list of power-products, we will say it is a *n-list* if it contains n power-products which are not simple. This fact extends what is called *variant* (A) in [2].

2.3.1. *0-Base-case*

Let I be generated by simple-powers, i.e., $(X_{n_1}^{a_{n_1}},\dots,X_{n_s}^{a_{n_s}})$. This is a 0-list. Note that $X_{n_1}^{a_{n_1}},\dots,X_{n_s}^{a_{n_s}}$ form a regular sequence. Repeating the same reasoning as in Example 3 and recalling from (4) that $\langle 0\rangle = 1$ we get

$$\langle X_{n_1}^{a_{n_1}},\dots,X_{n_s}^{a_{n_s}}\rangle = \prod_{i=1,\dots,s}\left(1-t^{\deg X_i^{a_{n_i}}}\right)\langle 0\rangle = \prod_{i=1,\dots,s}\left(1-t^{\deg X_i^{a_{n_i}}}\right) \tag{6}$$

which yields for the standard-case

$$\langle X_{n_1}^{a_{n_1}},\dots,X_{n_s}^{a_{n_s}}\rangle = \prod_{i=1,\dots,s}(1-t^{a_{n_i}}).$$

2.3.2. *1-Base-case*

Consider the 1-list $(\mathcal{P},X_{n_1}^{a_{n_1}},\dots,X_{n_s}^{a_{n_s}})$ where $\mathcal{P} = X_1^{p_1}\dots X_N^{p_N}$. Since we assume to work with interreduced lists (see Section 5.2), we have $a_{n_i} > p_{n_i}$ for each $i = 1,\dots,s$. Combining the results in (3) and (6) we have

$$\langle \mathcal{P},X_{n_1}^{a_{n_1}},\dots,X_{n_s}^{a_{n_s}}\rangle = \langle X_{n_1}^{a_{n_1}},\dots,X_{n_s}^{a_{n_s}}\rangle - t^{\deg\mathcal{P}}\langle X_{n_1}^{a_{n_1}} : \mathcal{P},\dots,X_{n_s}^{a_{n_s}} : \mathcal{P}\rangle$$

from which it follows that

$$\langle \mathcal{P},X_{n_1}^{a_{n_1}},\dots,X_{n_s}^{a_{n_s}}\rangle = \prod_{i=1,\dots,s}\left(1-t^{\deg X_{n_i}^{a_{n_i}}}\right) - t^{\deg\mathcal{P}}\left(\prod_{i=1,\dots,s}\left(1-t^{\deg X_{n_i}^{a_{n_i}-p_{n_i}}}\right)\right) \tag{7}$$

and for the standard-case

$$\langle \mathcal{P},X_{n_1}^{a_{n_1}},\dots,X_{n_s}^{a_{n_s}}\rangle = \prod_{i=1,\dots,s}(1-t^{a_{n_i}}) - t^{\deg\mathcal{P}}\left(\prod_{i=1,\dots,s}(1-t^{a_{n_i}-p_{n_i}})\right).$$

Note that if $\mathcal{P}$ is coprime with all the simple-powers $X_{n_i}^{a_{n_i}}$, then $p_{n_i} = 0$ for each $i = 1,\dots,s$; thus (7) becomes simpler:

$$\langle \mathcal{P},X_{n_1}^{a_{n_1}},\dots,X_{n_s}^{a_{n_s}}\rangle = \left(1-t^{\deg\mathcal{P}}\right)\left(\prod_{i=1,\dots,s}\left(1-t^{\deg X_{n_i}^{a_{n_i}}}\right)\right).$$

3. The algorithm

Input: $I = (T_1, \ldots, T_s)$ with T_i power-products in $A = k[X_1, \ldots, X_N]$
Output: $\langle I \rangle$

```
Function HPNum(I)
begin
    if $I$ is a base-case then return $\langle I \rangle$;
    else if $I$ is a splitting-case then return HPNum($I_1$) $\cdot \cdots \cdot$ HPNum($I_r$);
    else
        choose a pivot $\mathscr{P}$;
        return HPNum($I, \mathscr{P}$) + $t^{\deg \mathscr{P}}$HPNum($I : \mathscr{P}$);
end.
```

4. Examples

In this section we briefly mention the computational examples to compare the different strategies we have implemented and to give the timings we get with our version in CoCoA 3.

All the timings we give are referred to standard Hilbert–Poincaré series computations with 32-bit coefficients and assuming the input list interreduced.

If you use 32-bit integers, then $\langle X_1, \ldots, X_{34} \rangle = (1-t)^{34}$ cannot be computed because the coefficient of t^{17} in $(1-t)^{34}$ is $\binom{34}{17} = 2\,333\,606\,220$ and it is bigger than $2^{31} - 1 = 2\,147\,483\,647$ which is the biggest signed 32-bit integer. Note also that $\langle (X, Y)^D \rangle = 1 - (D+1)t^D + Dt^{D+1}$, and then there is no upper bound for the coefficients of the Hilbert–Poincaré numerator depending on the number of indeterminates. This means that no finite arithmetics can guarantee the result. The simplified Hilbert–Poincaré series of the examples we mention are computed correctly.

First of all we operate on some well-known benchmarks introduced by Bayer and Stillman in [2], and we refer the reader to that paper for their detailed description. They are listed in Table 1.

We will name "power N, D" the ideals generated by the first 10 000 power-products in lexicographic order of $\{X_1, \ldots, X_N\}^D$.

Table 1

Example	Indet's	PP's	Example nr.
mayr12	21	444	4.4
mayr13	21	610	4.4
prod4	32	500	4.2
square5	25	1371	4.1
mayr22	31	3204	4.4
mayr23	31	8100	4.4
prod5	50	4785	4.2

A new class of benchmarks, suggested by Jesus De Loera and Bernd Sturmfels, is represented by the "chess-examples". To compute them label with an indeterminate each square of an $n \times n$ chess-board. The ideal for a fixed piece is the ideal generated by the power-products X_iX_j where the move from the square labelled X_i to the square labelled X_j is a legal move for that piece. For example, the ideal "king 8×8" contains $X_{A1}X_{A2}$, $X_{A1}X_{B1}$, $X_{A1}X_{B2}$, …, and does not contain $X_{A1}X_{A3}$, $X_{A1}X_{G3}$. These are a subset of the graph-ideals, generated by the power-products of the couple of "vertices" of a graph linked with an edge. This class was used to prove that the problem of computing the dimension, a sub-problem of the computation of the Hilbert-Poincaré series, is NP-complete (see [2, Proposition 2.9], and also [8, 11]).

5. The pivot-case

5.1. Choosing the pivot

In Section 2 we said that all the recent algorithms are based on Eq. (3):

$$\langle I\rangle = \langle I,\mathscr{P}\rangle + t^{\deg \mathscr{P}}\langle I : \mathscr{P}\rangle.$$

Let us see a very small example in order to understand the importance of choosing a good pivot and highlight the behaviour of the different choices.

Example 4. Let I be the ideal generated by the 4-list

$$(xz^3, x^2y^2z, xy^3z, x^3yzw).$$

Let $\mathscr{P} = yw$:

$$\langle xz^3,\ x^2y^2z,\ xy^3z,\ x^3yzw\rangle = \langle xz^3,\ x^2y^2z,\ xy^3z,\ yw\rangle + t^2\langle xz^3,\ x^2yz,\ xy^2z,\ x^3z\rangle.$$

Let $\mathscr{P} = xz$:

$$\langle xz^3,\ x^2y^2z,\ xy^3z,\ x^3yzw\rangle = \langle xz\rangle + t^2\langle z^2,\ xy^2z,\ y^3,\ x^2yw\rangle.$$

The first choice generates two 4-lists (though simpler) and the second gives a 1-list and a 2-list. The latter is definitely better.

Let us have a closer look at the known algorithms.

5.1.1. Indeterminate-pivot

If the pivot $\mathscr{P}$ is an indeterminate, then $(I,\mathscr{P})$ is likely to have much fewer minimal generators than I because, as $\mathscr{P}$ is very small, many generators may be multiples of $\mathscr{P}$. On the other hand, $(I : \mathscr{P})$ might be as large as I is because of the same reason, especially if the power-input has high degree. Consider the ideal of Example 4 with $\mathscr{P} := x$. We get a 0-list and a 3-list:

$$\langle xz^3,\ x^2y^2z,\ xy^3z,\ x^3yzw\rangle = \langle x\rangle + t\langle z^3,\ xy^2z,\ y^3z,\ x^2yzw\rangle.$$

This is the choice suggested in [10]. Refs. [3, 12] follow the same idea, but split the computation into several ones whose input lists have no more occurrences of the pivot-indeterminate – these lists are defined using (3) inductively. This strategy is extremely sensitive to the choice of the indeterminate because it forces the pivot for several pivot-cases, but [3] suggests a good but costly procedure to look for the best one (in order to minimize the number of base-cases) and proves that the algorithm is optimal on Borel Fixed ideals.

5.1.2. *Generator-pivot*

In [2] the input ideal is regarded as $(I, \mathscr{P})$, where the pivot $\mathscr{P}$ is the least generator in RevLex; then (3) is read as $\langle I, \mathscr{P}\rangle = \langle I\rangle - t^{\deg \mathscr{P}}\langle I : \mathscr{P}\rangle$. Therefore they always get the ideal I which has exactly one generator less than the original input ideal, and the ideal $(I : \mathscr{P})$ which is expected to have very few minimal generators. In Example 4, $\mathscr{P} = x^3yzw$ gives a 3-list and a 0-list:

$$\langle xz^3,\ x^2y^2z,\ xy^3z,\ x^3yzw\rangle = \langle xz^3,\ x^2y^2z,\ xy^3z\rangle - t^6\langle z^2, y\rangle.$$

In particular, [2] shows that if the input ideal is Borel Fixed, then $(I : \mathscr{P})$ is a 0-base-case. Hence, the generator-pivot strategy is also theoretically optimal on Borel Fixed ideals.

Note that in both cases the computation splits into two branches, one far bigger than the other.

5.1.3. *GCD-pivot*

Ref. [4] points out that usually combinatorial algorithms can be speeded up by a “Divide and Conquer” approach, i.e., splitting the problem into two smaller problems of approximately the same size. Hence it suggests taking as pivot the GCD of three random power-products chosen among those containing the indeterminate occurring most. In Example 4 both x and z occur in each power-product. The possible GCDs in the list are xz or xyz. We have already seen that $\mathscr{P} := xz$ gives a 1-list and a 2-list. If we choose $\mathscr{P} := xyz$, then we get two 2-lists:

$$\langle xz^3,\ x^2y^2z,\ xy^3z,\ x^3yzw\rangle = \langle xyz,\ xz^3\rangle + t^3\langle z^2,\ xy,\ y^2,\ x^2z\rangle.$$

Being the choice of the GCD-pivot “random”, it cannot be proved to be “optimal” on some class of examples, but experience shows that it performs well on many types of inputs.

After what we have showed we would say that the GCD-pivot is the best strategy. It is better than the approach in [3] because its pivot selection is very fast. It is better than the indeterminate-pivot because it leads to fewer base-cases. It is better than the generator-pivot because the more balanced computation gives shorter interreductions of the lists $(I : \mathscr{P})$ and this is the most time-costly operation in the computation of Hilbert–Poincaré series.

5.1.4. *The new choice: simple-power-pivot*

Despite the fact that GCD-pivot strategy has a better behaviour than the indeterminate-pivot, the latter may give better timings. In fact, if the pivot is a simple-power, the reduction of $(I, \mathscr{P})$ is faster and there are nice tricks to discard a priori most of the divisibility tests normally needed while interreducing the list $(I : \mathscr{P})$. (See Section 5.2.)

Moreover, we are usually interested in computing the Hilbert–Poincaré series for **N**-graded algebras, especially standard algebras. In this case $\langle I \rangle$ is a univariate polynomial and the procedures operating on univariete polynomials can easily be optimized leading to very fast computation for the base-cases. (See Section 7.1.1.)

It follows that for **N**-graded algebras it might happen that the advantage given by the easier interreductions of the indeterminate-pivot approach may be bigger than the disadvantage of computing a larger number of base-cases.

Experience shows that most of the GCD-pivots are actually simple-powers after a few recursion steps and therefore the tricks are applicable to them, but the few multivariate-pivots lead to very costly interreductions and make this better choice slower in practice.

At this point the two winning ideas are a simple-power pivot to have faster interreductions and a "medium" pivot to have more balanced splittings. The compromise we suggest is a simple-power of the indeterminate occurring most, with exponent being that of this indeterminate in the GCD of two randomly chosen generators.

5.1.5. *Comparison of indeterminate-pivot, simple-power-pivot and GCD-pivot*

Table 2 shows that the GCD-pivot strategy gives fewer base and pivot-cases than the indeterminate-pivot and the simple-power-pivot but the best timings are usually given by the simple-power strategy.

The timings refer to the computation of the standard Hilbert–Poincaré series and it is easy to see that they are in general fairly similar. We give this detailed table to explain to the programmers the different behaviours and let them choose the best strategy for the class of examples they want to compute. For example, if the computation of the

Table 2

Example	Ind	Mon	Base-cases ind/SP/GCD	Pivot-cases ind/SP/GCD	Time ind/SP/GCD
mayr12	21	444	898/889/766	586/614/483	0.2/0.2/0.2
mayr13	21	610	1912/1404/1286	1242/921/752	0.5/0.4/0.4
*prod4	32	500	1113/1113/773	1036/1036/685	0.4/0.4/0.3
square5	25	1371	2033/1981/1686	1880/1824/1493	0.9/0.9/0.9
mayr22	31	3204	12 611/10 692/9194	8045/6766/5254	4.5/4.1/4.1
mayr23	31	8100	58 101/34 608/32 191	40 518/22 706/18 465	21.7/14.7/25.8
*prod5	50	4785	45 564/45 564/44 635	39 709/39 709/35 227	39.9/38.9/41.1
power 10, 50	10	10 000	3176/3095/3095	3175/3094/3095	7.1/5.5/8.2
power 20, 50	20	10 000	1529/1485/1485	1528/1484/1483	6.3/4.8/6.3
power 20, 20	20	10 000	1499/1485/1484	1498/1484/1483	5.2/4.7/6.3

base-cases is relatively slow, (e.g., multivariate series, integers with arbitrary precision, or computing small examples by hand) it is very important to minimize the number of the base-cases and then the best choice tends to be the GCD-pivot.

In the table there is an asterisk with prod4 and prod5; it is to point out those examples in which all the exponents are equal to one and then the simple-power-pivot and indeterminate-pivot strategies coincide. On the chess-examples, because of their particular structure, the three pivots are identical.

The examples *power* 20,20 and *power* 20,50 are essentially the same: in fact the power-products in *power* 20,50 are given by the power-products in *power* 20,20 multiplied by X_1^{30}. Comparing them shows how the indeterminate-pivot slows down if we raise the exponents, whereas the behaviours of the simple-power-pivot and the GCD-pivot do not change.

Notice also that the number of base-cases may be lower than the number of the power-products in the input list. This means that these strategies with our more general base-cases overcome the optimality on Borel Fixed ideals proved in [2, 3] for their algorithm.

5.2. *Reducing and dividing by the pivot*

After having chosen a pivot $\mathscr{P}$ for the ideal I the next step is to compute the two lists of the power-products generating the two ideals $(I, \mathscr{P})$ and $(I : \mathscr{P})$. We assume that the initial input is an interreduced list and we want to work on interreduced lists at every step. This choice is quite natural because a list is usually much smaller after being interreduced. Moreover, we shall see that interreducing the two lists is made considerably easier by knowing that the original list is interreduced.

In general, if we want to interreduce a list of power-products we have to remove all the redundant generators. In other words, for every power-product we must check all the other power-products in the list to see whether there is one dividing it. To reduce the number of divisibility tests we can order the list by decreasing degree and compare our power-product only with the following ones. Even so that number is very big: in fact if the list has n power-products the worst case gives $\frac{1}{2}n(n-1)$ comparisons – this happens for example if the list is already interreduced. However fast and optimized the divisibility tests between two power-products might be (see Section 7.2.1), the interreduction will take a very long time on a large input.

In our particular case, we know how the lists we need to interreduce were constructed and we can take advantage of this information to avoid a priori some divisibility tests. Let L be the minimal set of power-products generating I. A list of generators for $(I, \mathscr{P})$ is given by $L \cup \{\mathscr{P}\}$ and a list of generators for $(I : \mathscr{P})$ is given by $L' := (L : \mathscr{P}) = \{(T : \mathscr{P})\}_{T \in L}$ where $(T : \mathscr{P}) := \frac{T}{\mathrm{GCD}(T, \mathscr{P})}$.

5.2.1. *Interreducing $L \cup \{\mathscr{P}\}$*

From the fact that L is interreduced it is very easy to interreduce $L \cup \{\mathscr{P}\}$: in fact we only have to reduce L by $\mathscr{P}$, i.e., to delete from L all the power-products divisible by $\mathscr{P}$.

Table 3

Example	Ind	Mon	Only total-splits	$8 \sim N/2$	$4 \sim N/2$	$8 \sim N$
mayr12	21	444	0.2	0.2	0.3	0.3
mayr13	21	610	0.4	0.4	0.4	0.4
prod4	32	500	0.4	0.4	0.5	0.5
square5	25	1371	0.9	0.9	1.0	1.0
mayr22	31	3204	3.9	4.1	4.4	4.5
mayr23	31	8100	14.0	14.7	15.5	15.9
prod5	50	4785	37.8	38.9	41.6	43.6
knight 8×8	64	168	169.1	29.2	33.0	30.9
king 8×8	64	210	516.4	77.8	88.1	68.1
simple bad example	50	49	216.1	10.5	12.1	0.1

Again, these timings should help the programmers to choose according to the examples they want to compute. Checking short lists never seems to be useful ($4 \sim N/2$) and general splittings are not usually important if they are interested only in examples produced by a Gröbner basis computation.

Warning: even though the Hilbert–Poincaré series is correct, the coefficients of the Hilbert–Poincaré numerator of the chess examples grow larger than 32 bits.

7. Data structures

7.1. Polynomials

We have seen that we need only a few and very specialized operations on polynomials. Recall from Section 2 what we want to compute:

- For the base-cases

$$\langle \mathscr{P}, X_{n_1}^{a_{n_1}}, \dots, X_{n_s}^{a_{n_s}} \rangle = \prod_{i=1,\dots,s} \left(1 - t^{\deg X_{n_i}^{a_{n_i}}}\right) - t^{\deg \mathscr{P}} \left(\prod_{i=1,\dots,s} \left(1 - t^{\deg X_{n_i}^{a_{n_i} - p_{n_i}}}\right)\right).$$

- For the pivot-case

$$\langle I \rangle = \langle I, \mathscr{P} \rangle + t^{\deg \mathscr{P}} \langle I : \mathscr{P} \rangle.$$

- For the splitting-case

$$\langle I \rangle = \langle I_1 \rangle \dots \langle I_r \rangle.$$

Thus, if P, P_1 and P_2 are polynomials and D a (multi)degree, the only operations we need are:

$$(1 - t^D)P\ , \qquad P_1 - t^D P_2\ , \qquad P_1 + t^D P_2\ , \qquad P_1 \cdot P_2\ .$$

It is worth implementing optimised functions to compute them directly instead of using the generic arithmetic operations. In particular, the function which computes

$(1 - t^D)P$ is called many times for each base-case, thus it must be very efficient. Moreover, it can often be used in splitting-cases instead of $P_1 \cdot P_2$: in fact, when we split the list generating I, most of the I_j's have only one simple-power or one non-simple power-product and no simple-powers, thus their Hilbert–Poincaré numerator are of the form $(1 - t^D)$ and can be multiplied using $(1 - t^D)P$. It follows that there are very few "real" multiplications to compute, so the operation $P_1 \cdot P_2$ does not need to be particularly efficient.

7.1.1. Univariate polynomials

The numerator of the Hilbert–Poincaré series of an $\mathbf{N}$-graded ideal (either standard or weighted) is a univariate polynomial. To make the computation very fast in this case, we chose to represent univariate polynomials as vectors of integers, and implemented only the four operations described earlier.

The resulting polynomial actually tends to be "dense", i.e., with very few zero coefficients. This fact means that in our case the dense representation is faster and less space-costly than the sparse representation.

In the case of standard Hilbert–Poincaré series we can precompute a table of powers of $(1-t)$ thus saving many calls to the function computing $(1-t^D)P$: in fact, in many base-cases all the simple-powers have degree 1, so $\prod_{i=1,\dots,s}\left(1 - t^{\deg X_{n_i}^{a_{n_i}}}\right) = (1-t)^s$.

Note that a pre-computed table of this sort works only for the standard case and cannot be easily generalised for weighted or multivariate Hilbert–Poincaré series.

7.2. Power-products

The computation of the Hilbert–Poincaré series involves several calls of functions operating on power-products. In particular, among all the functions, the divisibility test between two power-products is the one called most often. Then it is very important to choose a good representation of power-products in order to get fast operations on them.

Let m be the power-product $X_1^{T_1} \dots X_N^{T_N}$. Using the dense representation, i.e., storing the exponents $T_1, \dots, T_N$ in a vector, we can access directly to the exponent of a given indeterminate. Using the sparse representation, i.e., listing the pairs $(n_1, T_{n_1}), \dots, (n_s, T_{n_s})$ with $T_{n_j} \neq 0$, we skip useless operations on zero exponents and, if implemented as a linked list, it is less space-costly when the number of indeterminates with non-zero exponent is small.

We decided to get advantage of all the information given by the two representations to make each operation as direct as possible. So our power-products are given by:

- Dense representation: a vector of integers with the exponents of all the indeterminates, $(T_1, \dots, T_N)$.
- Support: a vector of integers with the non-ordered list of the indices of the indeterminates with non-zero exponent $(n_1, \dots, n_s)$.
- Bit-support: a 32-bit unsigned integer representing part of $\mathrm{Supp}(m)$ that we shall describe shortly.

Note that in the case $\mathscr{P}$ is a simple-power, then the divisibility test with $\mathscr{P}$ is the simplest possible because we have to check only one exponent for each power-product in the list L.

5.2.2. *Interreducing L'*

Recall that L is interreduced, i.e., for each $T_a = X_1^{a_1} \dots X_N^{a_N}$ and $T_b = X_1^{b_1} \dots X_N^{b_N}$ if $T_a \neq T_b$, then T_a does not divide T_b and hence there exists an i such that $a_i > b_i$.

Consider the case of a simple-power-pivot $\mathscr{P} = X_p^d$. There are three interesting remarks:

- If $a_p \leq b_p$, then $i \neq p$ and $(T_a : \mathscr{P})$ cannot divide $(T_b : \mathscr{P})$ because we still have $a_i > b_i$ in these two power-products.
- If $a_p > d$, $b_p > d$, then $(T_a : \mathscr{P})$ cannot divide $(T_b : \mathscr{P})$ because we still have $a_i > b_i$ in these two power-products: if $i \neq p$ then $a_i > b_i$, and if $i = p$ then $a_p - d > b_p - d$.
- If $a_p > d \geq b_p$, then $(T_a : \mathscr{P})$ cannot divide $(T_b : \mathscr{P})$ because the exponent of X_p in $(T_a : \mathscr{P})$ is positive whereas in $(T_b : \mathscr{P})$ it is 0.

In summary:

Proposition 1. *Define*

$$L[r] := \{T_a \in L \mid a_p = r\} \quad \textit{for } r = 0, \dots, d;$$

$$L[d+1] := \{T_a \in L \mid a_p > d\},$$

then

1. *$L'[s] := (L[s] : \mathscr{P})$ is interreduced for each $s = 0, \dots, d+1$;*
2. *a power-product in $L'[s]$ may divide some power-product in $L'[r]$ only if $r < s \leq d$.*

These remarks drastically reduce the number of divisibility tests, but they do not easily generalise for non-simple-power pivots, not even for power-products with only two indeterminates. Take, for example, $\mathscr{P} = x^3y^3$, $T_a = xy^3z^a$ and $T_b = x^2z^b$: then $(T_a : \mathscr{P}) = z^a$ and $(T_b : \mathscr{P}) = z^b$, so it is clear that nothing can obviously be said a priori about which may divide which knowing only the degrees of the "pivot-indeterminates".

But there is a natural extension for $L[0]$ and $L[d+1]$: the first, L_{c}, given by the power-products coprime with the pivot, the second, L_{bm}, given by the "big-multiples" of the pivot, where T_a is a **big-multiple** of T_b if $a_i > b_i$ for every $b_i \neq 0$; with similar reasoning we get:

Proposition 2. 1. *$(L_{\mathrm{c}} : \mathscr{P})$ and $(L_{\mathrm{bm}} : \mathscr{P})$ are interreduced;*

2. *the elements of $(L_{\mathrm{c}} : \mathscr{P})$ divide no other power-product in L';*

3. *the elements of $(L_{\mathrm{bm}} : \mathscr{P})$ neither divide nor are divided by any other power-product in L'.*

It follows that the bigger these subsets are, the fewer divisibility tests are needed in order to interreduce L'; in other words, the smaller the pivot, the faster the interreduction.

6. The splitting-case

Recall from the definition of splitting-case (Section 2.2) that it occurs when we can partition the set of the indeterminates into disjoint non-empty subsets $\mathbf{X}_1,\dots,\mathbf{X}_s$ such that each generator of our monomial ideal I belongs to a $k[\mathbf{X}_j]$ for some j; let $I_j := I \cap k[\mathbf{X}_j]$. In this case, the Hilbert–Poincaré series can be computed via (5):

$$\langle I \rangle = \langle I_1 \rangle \dots \langle I_r \rangle.$$

Let L be the n-list generating I. If n is large, checking if such a partition exists might be very costly. Moreover, in general, its existence is quite unlikely when n is much larger than N, the number of indeterminates, and if it exists it often leads to splitting L into a 1-list and an $(n-1)$-list, a splitting which does not improve much the behaviour of the computation.

Definition 1. If a splitting-case occurs when the list "has become" very short after some pivot-cases, then usually the non-simple power-products are pairwise coprime. We call this special situation *total-splitting-case.*

The total-splitting-cases are far faster to spot: in fact, to compute the pivot we need to look for the indeterminate occurring most, so we simply have to check if it occurs only once to know whether it is a total-splitting-case or not. Furthermore, they are computed more directly than the splitting-cases because we know a priori that the lists generating $I_1,\dots,I_r$ are at most 1-lists.

After these remarks it seems it is not worth considering the general splitting-case. But the computation of some Hilbert–Poincaré series might be much faster if we look for them: recall the "Chess Examples" (Section 4) or the most dramatic "Simple Bad Example" described in [4], generated by

$$(X_0X_1,\ X_1X_2,\dots,X_{N-1}X_N).$$

Our experience suggested this compromise: we look for the possible partition only when $8 \le n \le N/2$. We exclude lists longer than $N/2$ because checking them is slow and very likely useless; we exclude $n < 8$ because such a short list would be more quickly computed via pivot-cases and total-splitting-cases.

Table 3 shows that this choice might slow down a little bit the computation of some examples whereas for some others the improvement is evident. For completeness we show also the timings obtained looking for general splitting-cases when $4 \le n \le N/2$ and when $8 \le n \le N$.

This way of representing power-products looks too space-costly, but in general the input list for our algorithm is the output of the computation of a Gröbner basis. So you can assume that if you have enough memory to compute a Gröbner basis with such an output, then you have enough memory to compute its Hilbert–Poincaré series using this redundant representation.

7.2.1. *Bit-support*

Given two power-products m and m', if Supp(m) is not contained in Supp(m'), then m cannot divide m'. When the number of indeterminates is large, power-products tend to have several zero exponents. Thus, checking the supports often tells us when the divisibility test would fail.

We represent $\mathrm{Supp}(m) = \{X_{i_1},\dots,X_{i_s}\}$ by the 32-bit unsigned integer whose binary expansions have 1s in the i_j-th places and 0 elsewhere, i.e.,

$$\sum_{j=1}^{s} 2^{(i_j-1)}.$$

This representation, which takes very little memory, allows us to compare the supports very quickly via built-in bitwise functions (we thank Thomas Yan for suggesting this technique):

Example 5. $\mathrm{Supp}(X_1^4X_3^2X_6) = \{X_1,X_3,X_6\}$ corresponds to $1+2^2+2^5 = 37 = 100101_2$. $\mathrm{Supp}(X_1X_3^2X_8) = \{X_1,X_3,X_8\}$ corresponds to $1+2^2+2^7 = 133 = 10000101_2$. Bitwise-AND gives $101_2 = 5 \neq 37$, i.e., the intersection of the two supports is $\{X_1,X_3\}$ which is strictly contained in $\{X_1,X_3,X_6\}$, and this tells us that the first power-product cannot divide the second. Moreover, note that $(\mathrm{Supp}(m) \cap \mathrm{Supp}(m')) = \emptyset$, i.e., (bit-Supp($m$) AND bit-Supp($m'$)) $= 0$, if m and m' are coprime.

Obviously, a 32-bit integer can represent only 32 indeterminates and obviously this sort of representation is not forced to use 32 bits. We decided to use “long” integers because they are the largest allowed by ANSI C, and they usually take 32 bits.

We tried some tests with more than 32 indeterminates keeping a list of integers in order to have a complete representation for the support, but the advantage we obtained by this more general structure is smaller than the cost of managing it. However, notice that such a list is certainly a good representation of square free power-products for algorithms computing the dimension.

7.2.2. *Divisibility test*

Let m and m' be two power-products. The *divisibility test* is the function saying whether m divides m'. It is computed checking whether the bit-support of m is contained in the bit-support of m'; if it is, then a *full divisibility test* is computed checking the exponents in m and in m' of the indeterminates in the support of m.

Table 4

Example	Ind	Mon	DT	|Supp(m)| DT	bit-Supp(m) DT
mayr12	21	444	61 874	41 712	2 003
mayr13	21	610	83 674	56 004	3 031
prod4	32	500	187 570	159 071	2 951
square5	25	1371	785 670	508 723	7 952
mayr22	31	3204	1 639 522	1 002 749	25 734
mayr23	31	8100	5 289 321	2 705 617	91 373
prod5	50	4785	22 912 018	19 755 624	456 413

The "DT" column in Table 4 gives the number of divisibility tests we need to compute the Hilbert–Poincaré series and the "bit-Supp(m) DT" column shows the number of full divisibility tests after checking the bit-support.

In order to emphasize the importance of the information given by the bit-support, we show in column "|Supp(m)| DT" how many full divisibility tests we would have to perform after having verified only that the cardinality of the support of m is smaller than the cardinality of the support of m'.

8. Timings

Table 5 compares the timings given by our implementation in CoCoA 3 (currently in α-version) running on a 50 MHz Sun Microsystems Sparc Station 10, and compiled with gcc 2.5.6, and the best timings published in [2]. We recall that they used a slower Sparc Station 1.

The two timings given in the CoCoA column, refer to the algorithm computing time and the complete *Poincare* function time, i.e., they differ only by the time spent in reading and storing the input file.

We remind that the timings we give are referred to standard Hilbert–Poincaré series computations with 32-bit coefficients (all these Hilbert–Poincaré numerators are computed correctly) and assuming the input list interreduced.

Table 5

Example	Ind	Mon	CoCoA 3	B&S
mayr12	21	444	0.24/0.36	21
mayr13	21	610	0.37/0.53	45
prod4	32	500	0.44/0.55	35
square5	25	1371	0.91/1.17	186
mayr22	31	3204	4.11/4.86	3056
mayr23	31	8100	14.67/16.65	22013
prod5	50	4785	38.88/39.77	10403

Acknowledgements

Sincere thanks go to L. Robbiano, T. Yan and J. Hollman for their useful remarks and suggestions.

References

[1] M.F. Atiyah and I.G. MacDonald, Introduction to Commutative Algebra, Addison-Wesley Series in Mathematics (Addison-Wesley, Reading, MA, 1969).

[2] D. Bayer and M. Stillman, Computation of Hilbert functions, J. Symbol. Comput. 14 (1992).

[3] A.M. Bigatti, M. Caboara and L. Robbiano, On the computation of Hilbert–Poincaré Series, AAECC J. 2 (1991).

[4] A.M. Bigatti, P. Conti, R. Robbiano and C. Traverso, A "Divide and Conquer" algorithm for Hilbert–Poincaré series multiplicity and dimension of monomial ideals, Proc. of AAECC-10, Lecture Notes in Computer Science, Vol. 673 (Springer, NewYork, 1993).

[5] B. Buchberger, Gröbner bases: an algorithmic method in polynomial ideal theory, in: N.K. Bose, ed., Recent Trends in Multidimensional Systems Theory, chap. 6 (Reidel, New York, 1985) 184–232.

[6] M. Caboara, G. De Dominicis and L. Robbiano, Multigraded Hilbert functions and Buchberger algorithm, Proc. ISSAC 96, to appear.

[7] A. Capani, G. Niesi and L. Robbiano, CoCoA, a system for doing Computations in Commutative Algebra, Available via anonymous ftp from `lancelot.dima.unige.it` (1995).

[8] M. Garey and D. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness (Freeman, San Francisco, 1979).

[9] P. Gianni, T. Mora, L. Robbiano and C. Traverso, Hilbert functions and Buchberger algorithm, Preprint (1993).

[10] J. Hollman, On the computation of the Hilbert series, Latin 92, São Paulo, Lecture Notes in Computer Science, Vol. 583 (Springer, New York, 1992).

[11] R.M. Karp, Reducibility among combinatorial problems, in: R.E. Miller and J.W. Thatcher, eds., Complexity and Computer Computations (Plenum Press, New York, 1972) 85–103.

[12] M.V. Kondrat'eva and E.V. Pankrat'ev, A recursive algorithm for the computation of Hilbert polynomial, EUROCAL 87, Lecture Notes in Computer Science, Vol. 387 (Springer, New York, 1987).

[13] F.S. Macaulay, Some properties of enumeration in the theory of modular systems, Proc. London Math. Soc. 26(2) (1927) 531–555.

[14] M. Möller and T. Mora The computation of the Hilbert function, EUROCAL 83, Lecture Notes in Computer Science, Vol. 162 (Springer, New York, 1983).

[15] R.P. Stanley, Hilbert functions of graded algebras, Adv. Math. 28 (1978) 57–83.