A Note on Maple's GCD Computation of Partially Factored Polynomials

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria
Wolfgang.Schreiner@risc.uni-linz.ac.at

April 9, 2000

Abstract

We report on a performance anomaly that may occur when Maple computes the greatest common divisor of partially factored polynomials, i.e., polynomials in the "product of factors" representation. Maple utilizes the partial factorization of a polynomial to compute the gcd from the individual factors. We have encountered an application in algebraic geometry where this "optimization" makes the gcd computation very slow which also affects the overall performance. We consider this as a hint that the view of a polynomial as a syntactic expression is a source of pitfalls.

1 The Maple Function

Polynomials In Maple (Version V Release 5.1), a polynomial is a symbolic expression constructed from names (unassigned variables) and numerical values using the arithmetic operators $+$, $-$, $*$ and $\hat{}$, e.g. the definitions

$$p := x^2 + x;$$
$$q := x(x+1)$$

introduce two different polynomials p and q (no canonical simplification takes place that could identify them). The predicates `type(`a`,‘+‘)` and `type(`a`,‘*‘)` test whether a polynomial a is a sum of terms or a product of factors, i.e., both `type(`p`,‘+‘)` and `type(`q`,‘*‘)` return true.

GCD of Polynomials The Maple function `gcd(`a`,`b`,’`*cofa*`’,’`*cofb*`’)` returns the greatest common divisor g of two polynomials a and b and sets the variables *cofa* and *cofb* to the cofactors a/g and b/g respectively (see Figure 1).

If a and b are both polynomials with integer coefficients or polynomials with rational coefficients, then the `gcd` function tests whether a is a product of sums or b is a product of sums and, if the test succeeds, delegates the task to the function `‘gcd/PartFactoredCase‘` (with the variable Z set to true iff a and b have integer coefficients).

```
gcd := proc(aa, bb, cofa::name, cofb::name)
local Z, GCD, a, b, t1, fb, fa, u, v;
option
`Copyright (c) 1992 by the University of Waterloo. All rights reserved.`;
...
    if (type(aa, `*`) or type(aa, `^`)) and hastype(aa, `+`) or
    (type(bb, `*`) or type(bb, `^`)) and hastype(bb, `+`) then
        if aa = 0 then GCD := bb
        elif bb = 0 then GCD := aa
        else GCD := `gcd/PartFactoredCase`(aa, bb, Z, args[3 .. nargs])
        fi;
...
end:
```

Figure 1: Maple's `gcd` function

GCD of Partially Factored Polynomials The `` `gcd/PartFactoredCase` `` function utilizes the product of factors representation of a respectively b by constructing two arrays

$$la := \texttt{factorlist}(a)$$
$$lb := \texttt{factorlist}(b)$$

where $\texttt{factorlist}(p)$ is an array of the factors of p. Each factor is represented by a pair $(term, exp)$ where $term$ is the base term of the factor and exp is its exponent. For instance, the polynomials

$$x(x+1)$$
$$x^2(x+1)$$
$$x^2+1$$

are converted to the arrays

$$[(x,1),(x+1,1)]$$
$$[(x,2),(x+1,1)]$$
$$[(x^2+1,1)]$$

respectively.

The basic idea of the program is to compute the gcd g of a and b by computing the gcds of all factors of a and of all factors of b and to combine the results to g (see Figure 2).

In more detail, the function proceeds by iterating over the arrays la and lb to compute two arrays res and $mult$ while preserving, in iteration i of the outer loop and iteration j of the inner loop, the invariants

$$\forall i' < i : \forall j' < j : \text{mono}(la[i']) \text{ and } \text{mono}(lb[j']) \text{ are squarefree}$$
$$a = \text{poly}(la) * \textstyle\prod_k res[k]^{mult[k]}$$
$$b = \text{poly}(lb) * \textstyle\prod_k res[k]^{mult[k]}$$

where $\text{poly}(lp) := \prod_k \text{mono}(lp[k])$ and $\text{mono}(lm) := lm[1]^{lm[2]}$. In other words, every iteration i/j extracts the common factor of $la[i]$ and $lb[j]$ into the arrays res (the base value) and $mult$ (the exponent).

```
`gcd/PartFactoredCase` := proc(aa, bb, Z, cofa, cofb)
local factorlist, laa, lbb, naa, nbb, k, i, j, t, res, nul, ca, cb, mult;
option `Copyright (c) 1995 by Waterloo Maple Inc. All rights reserved.`;
    factorlist := proc(aa)
        local i;
        option
        `Copyright (c) 1995 by Waterloo Maple Inc. All rights reserved.`;
            if type(aa, `*`) then [seq(
                `if`(i::`^`, [expand(op(1,i)), op(2,i)], [expand(i),1])
                , i = aa)]
            elif type(aa, `^`) then [[expand(op(1, aa)), op(2, aa)]]
            else [[expand(aa), 1]]
            fi
        end;
    laa := factorlist(aa);
    lbb := factorlist(bb);
    naa := nops(laa);
    nbb := nops(lbb);
    laa := table(laa);
    lbb := table(lbb);
    k := 0;
    for i while i <= naa do
      for j while j <= nbb and not type(laa[i][1], integer) do
            if type(lbb[j][1], integer) then next fi;
            t := `gcd/doit`(laa[i][1], lbb[j][1], Z);
            if t <> 1 then
                divide(laa[i][1], t, 'ca');
                divide(lbb[j][1], t, 'cb');
                k := k + 1;
                res[k] := t;
                mult[k] := min(laa[i][2], lbb[j][2]);
                lbb[j] := [cb, lbb[j][2]];
                laa[i] := [ca, laa[i][2]];
                if mult[k] < lbb[j][2] then
                    nbb := nbb + 1; lbb[nbb] := [t, lbb[j][2] - mult[k]]
                elif mult[k] < laa[i][2] then
                    naa := naa + 1; laa[naa] := [t, laa[i][2] - mult[k]]
                fi
            fi
        od
    od;
    if 3 < nargs then
        cofa := `*`(seq(laa[i][1]^laa[i][2], i = 1 .. naa));
        if 4 < nargs then
            cofb := `*`(seq(lbb[i][1]^lbb[i][2], i = 1 .. nbb))
        fi
    fi;
    `*`(seq(res[i]^mult[i], i = 1 .. k))
end:
```

Figure 2: Maple's `gcd/PartFactoredCase` Function

As a consequence, we have after the termination of the loop

$$\begin{aligned}\gcd(a, b) &= \textstyle\prod_k \mathit{res}[k]^{\mathit{mult}[k]}\\ a &= \mathrm{poly}(la) * \gcd(a, b)\\ b &= \mathrm{poly}(la) * \gcd(a, b)\end{aligned}$$

i.e., *res* and *mult* hold the gcd of a and b and la and lb hold the cofactors.

Let us consider in more detail iteration i of the outer loop and iteration j of the inner loop and let $(d', e'') := la[i]$ and $(d'', e'') := lb[j]$. First, we compute the gcd g of d' and d''. If g is 1, the factors are squarefree and we can proceed to the next iteration.

If g is not one, we divide both d' and d'' by g and subtract $e := \min(e', e'')$ from both e' and e''. To maintain the first part of the invariant, it now suffices to append g to *res*, to append e to *mult*, and, if $e' > e$, to append $(g, e' - e)$ to la or, if $e'' > e$, to append $(g, e'' - e)$ to lb, respectively.

In subsequent loop iterations, we will profit from the reduced values of $la[i]$ and $lb[j]$ (which allows to compute the corresponding gcds faster). On the other hand, we will have to consider an additional factor appended to the end of la or of lb.

Performance The application of `` `gcd/PartFactoredCase` `` in Maple's `gcd` function is based on the tacit assumption that it is desirable to utilize the available factorization (i.e., the product of factors representation) of a polynomial, i.e., that `` `gcd/PartFactoredCase` `` returns a result faster than computing the gcd of the expanded polynomial (i.e., in a sum of terms representation). In the following section, we will demonstrate that this assumption is in general not true by giving a practical application in the context of which `` `gcd/PartFactoredCase` `` is much slower.

2 The Performance Anomaly

Application Context Maple provides the package `algcurves` with various functions to deal with algebraic curves. One such function is `singularities(`f`, `x`, `y`)` with returns the set of singularities of the curve defined by the squarefree polynomial f in variables x and y.

We investigated the dynamic behavior of this function by benchmarks that applied this function to various polynomials of larger degree (35 or so). Here we noticed that very frequently a good deal of the computation time (e.g. 9.2s in a 24.2s run, i.e. 38%) is spent in a single `gcd` computation, more concrete in a single application of `gcd` which in turn invokes `` `gcd/PartFactoredCase` ``.

As it turns out, this long-running `gcd` application has the form

```
gcd(a, diff(a, x))
```

within the function `` `sqrfree/Yun`(a,x) `` which in turn receives its argument from the call chain

```
singularities(f, x, y)
   ...
   F := ... f ...
   `algcurves/find_points`(F, x, y, ...)
`algcurves/find_points`(f, x, y, ...)
   ...
   j := discrim(f, y)
   `algcurves/db_factors`(j, x, ...)
`algcurves/db_factors`(f, x, ...)
   ...
   sqrfree(f, x, ...)
sqrfree(a, x)
   ...
   `sqrfree/sqrfree`(primpart(a, ...), x)
`sqrfree/sqrfree`(a, x)
   ...
   content(a, x, p)
   `sqrfree/Yun`(p, x)
`sqrfree/Yun`(a, x)
   ...
   gcd(a, diff(a, x))
```

Neglecting some minor details, the argument a to the long-running gcd computation is thus the discriminant j of a polynomial F that is essentially the expanded version of f.

As it turns out, the result of the `discrim` call is frequently (more often than not) of the form

$$x^n p$$

for variable x, integer n and polynomial p. The exponent n can be very high, e.g., in the case of some polynomial a of degree 35, n equals 123.

The application `gcd(`a`, diff(`a`, `x`))` thus has the form

$$\texttt{gcd(}x^n p\texttt{, } n x^{n-1} p + x^n p'\texttt{)}$$

where $p' := \frac{\partial p}{\partial x}$. If p is a sum of terms, the first argument is of the form "product of sums", and `gcd` consequently invokes `` `gcd/PartFactoredCase` ``.

Dynamic Behavior As shown in the first section, `` `gcd/PartFactoredCase` `` converts both arguments to the lists

$$\begin{aligned} la &:= [(x, n), (p, 1)] \\ lb &:= [(q_0, 1)] \end{aligned}$$

where $q_0 := n x^{n-1} p + x^n p'$.

It then starts the iteration $i = 1$ by computing the gcd of x and q which (if $n > 1$) yields x. The lists are correspondingly updated and yield

$$\begin{aligned} la &:= [(1, n), (p, 1), (x, n-1)] \\ lb &:= [(q_1, 1)] \end{aligned}$$

where $q_1 := nx^{n-2}p + x^{n-1}p'$.

The next but one iteration $i = 3$ repeats this pattern with the factors $(x, n-1)$ and $(q_1, 1)$ appending another factor $(x, n-2)$ to la. This action is repeated in some subsequent iteration yielding (possibly among others) $n - 1$ iterations in each of which the gcd of x and of $q_i := nx^{n-i-1} + x^{n-i}p'$ is computed (by a trial division). The outer loop thus has at least n iterations each of which involves some significant execution overhead.

Analysis The source of the anomaly is the fact that the first argument in the call of ‘`gcd/PartFactoredCase`‘ is partially factored while the second is in an expanded form.

If the first argument $x^n p$ was expanded to

$$\sum_i x^n p_i$$

`gcd` would not attempt to make use of the partial factorization but use another method (e.g. the modular algorithm) for gcd computation.

On the other hand, if the second argument was partially factored to

$$x^{n-1}(np + xp')$$

then ‘`gcd/PartFactoredCase`‘ would start with the factor lists

$$\begin{aligned} la &:= [(x, n), (p, 1)] \\ lb &:= [(x, n-1), (np + xp', 1)] \end{aligned}$$

which would after one iteration become

$$\begin{aligned} la &:= [(1, n), (p, 1), (x, 1)] \\ lb &:= [(1, n-1), (np + xp', 1)] \end{aligned}$$

from which the complete gcd could be computed very quickly (because the large exponent n of x hase disappeared and the constant factor 1 is ignored).

However, the combination of a partial factorization of one argument with the expanded version of the other argument both of which have a non-trivial common factor with a high exponent leads to the worst-case situation: we decrease the exponent of the common factor in every iteration by one which results in a large number of iterations.

3 Benchmarks

To demonstrate the (in)efficiency of `gcd/PartFactoredCase` in the context demonstrated above, we compare the original application of this function to two applications of `gcd` with modified arguments:

- one where the first argument is expanded to a sum of terms representation (such that another, presumably modular, gcd algorithm is applied);
- one where the second argument is factored to a product of terms representation (such that `gcd/PartFactoredCase` terminates quickly).

Below we give the the execution times for the various gcd computations (measured on a PC with a Celeron@500Mhz processor). The first column refers to the degree of the polynomial defining the algebraic curve that triggers the gcd computation (not to the degree of the polynomials whose gcd is actually computed).

`gcd`			
Degree	Original	Expanded	Factored
27	3.9s	0.2s	0.1s
29	11.6s	2.2s	0.2s
32	18.3s	0.3s	0.2s
33	19.6s	0.3s	0.1s
35	9.2s	0.2s	0.1s
37	22.9s	3.4s	0.6s
40	10.7s	1.1s	0.6s

We see that the expanded version is in call cases significantly (from 5 up to 65 times) faster than the original version. The factored version is even faster but requires a somewhat cumbersome computation of the derivative without the use of Maple `diff` function (because an afterwards factorization of the expanded derivative would be too costly).

For the computation of the singularities of an algebraic curve with Maple's `singularities` function, the overall performance gains of replacing the original gcd call by a call with the expanded versions of the polynomials can be significantly.

`algcurves`			
Degree	Original	Expanded	Speedup
27	8.9s	5.2s	1.7
29	28.0s	18.6s	1.5
32	31.4s	13.4s	2.3
33	34.9s	15.6s	2.2
35	24.2s	15.2s	1.6
37	203.0s	183.5s	1.1
40	194.2s	184.6s	1.1

However, we also see that for large examples the other costs of the `algcurves` function outweigh the performance gains.

4 Conclusions

As this paper demonstrates, it may be dangerous to rely on the syntactic form of an algebraic expression for optimizing the performance of a computation. The presented performance anomaly does not happen in systems like SACLIB that compute only with canonical representations of polynomials (e.g. as sequences of coefficient/exponents pairs). Considering polynomials just as symbolic expressions (as Maple does) is a source of pitfalls: it requires the programmer to deal with various syntactic cases and tempts to questionable optimizations.