Evaluation of the Heuristic Polynomial GCD

Hsin-Chao (Phil) Liao
Richard J. Fateman*
Computer Science Division, EECS Department
University of California at Berkeley

Abstract

The Heuristic Polynomial GCD procedure (GCDHEU) is used by the Maple computer algebra system, but no other. Because Maple has an especially efficient kernel that provides fast integer arithmetic, but a relatively slower interpreter for non-kernel code, the GCDHEU routine is especially effective in that it moves much of the computation into "bignum" arithmetic and hence executes primarily in the kernel.

We speculated that in other computer algebra systems an implementation GCDHEU would not be advantageous. In particular, if *all* the system code is compiled to run at "full speed" in a (presumably more bulky) kernel that is entirely written in C or compiled Lisp, then there would seem to be no point in recasting the polynomial gcd problem into a bignum GCD problem. Manipulating polynomials that are vectors of coefficients would seem to be equivalent computationally to manipulating vectors of big digits.

Yet our evidence suggests that one can take advantage of the GCDHEU in a Lisp system as well. Given a good implementation of bignums, for most small problems and many large ones, a substantial speedup can be obtained by the appropriate choice of GCD algorithm, including often enough, the GCDHEU approach. Another major winner seem to be the subresultant polynomial remainder sequence algorithm. Because more sophisticated sparse algorithms are relatively slow on small problems and only occasionally emerge as superior (on larger problems) it seems the choice of a fast GCD algorithm is tricky.

1 Introduction and objectives

It is well known that GCD computations are very important in computer algebra systems, especially in simplifying rational expressions, computing partial fraction expansions, and similar canonical transformations [5, 10]. It is also useful in constructing reduced characteristic sets to prove geometry theorems [2].

*This work was supported in part by NSF Grant number CCR-9214963 and by NSF Infrastructure Grant number CDA-8722788.

Some GCD algorithms are based on some variation of the Euclidean algorithm extended to operate over a polynomial ring. The most efficient known variant is the Subresultant Polynomial Remainder Sequence (PRS) algorithm [1, 3, 5, 6].

There are algorithms based on modular homomorphisms and their inverse mappings. Some algorithms require more than one homomorphism in the construction of the GCD. One example is the modular algorithm based on Chinese remaindering (GCDCHREM) used in Maple [5]. The algorithms requiring only one homomorphic mapping are usually based on the Hensel construction. Extended Zassenhaus GCD (EZGCD) and Extended EZGCD (EEZGCD) are of this type [8, 9]. Zippel's Sparse Modular algorithm (SPMOD) incorporates evaluation homomorphism in finite fields, probabilistic sparse interpolation and univariate GCD [10].

Rather different in nature, the Heuristic GCD algorithm (GCDHEU) uses evaluation homomorphism over the integers rather than finite fields [5]. It is not useful for large problems, and a carefully designed program will identify a failing case relatively rapidly, and move on to an alternative algorithm.

Of all the commercially available computer algebra systems, only Maple uses GCDHEU. Maple has a small and efficient kernel that highly optimizes arithmetic of integers, and has special code for univariate polynomials. More complicated computations such as multivariate polynomial GCDs, are programmed for execution by an efficient interpreter which is however much slower than kernel code. By mapping polynomials into integers, GCDHEU executes primarily in the kernel, simultaneously avoiding interpreter overhead in this important algorithm, and still not taking up additional kernel code space.

GCDHEU is not implemented in any other system, probably because their vendors assumed that in an environment where all GCD procedures are compiled at full speed in C or Lisp, there is no advantage in mapping polynomials (which are vectors of coefficients) into bignums (which are vectors of bigdigits). Rather than concentrating on implementation tricks, the tendency has been to seek algorithmic advantages (better asymptotic running times). The modular algorithm, EZGCD and SPMOD replace bignum arithmetic with allegedly faster modular single-word arithmetic. These minimize coefficient growth, where such growth is usually a problem. GCDHEU, on the other hand, requires much larger evaluation points, thus almost inevitably forces calculations to continue into the bignum regime. Since the size of evaluation point grows with the degree of the inputs, GCDHEU is becomes less effective for large polynomials.

Despite these drawbacks, our experiments suggest that GCDHEU is useful, and the compiled/kernel versus interpreter environment issue is not really a dominant factor. It can be used to good effect in a compiled Lisp environment. Our data show that, for low-degree polynomials in up to ten variables, GCDHEU is faster than the Subresultant PRS method in the cases where the GCDs are non-monic and dense. It is faster than the EZGCD and SPMOD in several cases, especially for polynomials with five or fewer variables.

2 Review of GCD algorithms

In this section, we briefly summarize each GCD algorithm, its advantages and disadvantages, and provide references.

Subresultant PRS: The original PRS algorithm is the Euclidean algorithm using pseudo-division in a polynomial ring. Its main problem is the large coefficient growth induced by pseudo-division. In the multivariate case, the coefficients are themselves polynomials, and coefficient growth includes an exponential growth in the degrees of the coefficients. The Primitive PRS algorithm decreases such growth to a minimum by repeatedly removing the content of the pseudo-remainders at each iteration of the PRS. Because the contents are computed by GCDs, the Primitive PRS requires many more operations than the Euclidean PRS.

The Subresultant PRS is a compromise between the Primitive and Euclidean. At each iteration, the pseudo-remainder is reduced by a factor that can be computed much more efficiently than the content. While the coefficients in the Subresultant PRS are sometimes not minimal, it eliminates recursive calls to the GCD operation. Despite the optimizations in Subresultant algorithms, the coefficients (which can be polynomials) still grow very large. An earlier "improved" PRS described in the literature, the reduced PRS, uses a different technique that is not as effective as the subresultant, but is hardly any cheaper, and so it is not advisable [3, 5, 6, 10].

Modular Algorithm using Chinese Remaindering: (GCDCHREM) The idea in modular algorithms is to eliminate growth in coefficient sizes by computing via evaluation homomorphisms to reduce multivariate problems to univariate ones, and to compute in finite fields to eliminate coefficient growth. More specifically, one can map the input polynomials into images in several finite fields, compute their GCDs in these fields and reconstruct the GCD of the original inputs from these "images" using a generalized Garner's algorithm (or Chinese Remainder Algorithm). Multivariate polynomial GCD problems are reduced to univariate problems by using (repeatedly) evaluation homomorphisms to eliminate variables.

In the case of a small GCD (say 1), the answer is rapidly uncovered, but a typical implementation of this algorithm is prepared to compute many evaluation and modular homomorphisms if a non-trivial GCD must be constructed. Also, the algorithms must detect and recover from the improbable, but not impossible, cases where the finite fields are "unlucky" resulting in incorrect modular GCDs. These unlucky homomorphisms are eventually discovered in the course of the calculation, however.

The drawback of this algorithm is that the number of homomorphisms needed is exponential in the number of variables [5].

Hensel Algorithms: EZGCD and EEZGCD: EZGCD uses Hensel lifting to reduce the number of homomorphisms. In a nutshell, the two input polynomials are reduced to two univariate polynomials whose GCD is then lifted back to the multivariate domain using a generalized Newton's iteration. In most cases, we only need to choose one or two values for each variable, resulting in much faster execution over the GCDCHREM algorithm. Another advantage of the Hensel construction is that in many cases, by choosing the right evaluation points, the sparsity of the input polynomials is preserved in the homomorphisms.

There are a few problems with this method. Hensel lifting often has trouble in computing the leading coefficients of non-monic GCDs correctly. Special handling in these cases drastically hampers performance. And secondly, Hensel lifting requires that the homomorphic GCD be relatively prime to one of its cofactors. Bad evaluation points or inputs can results in the case where a GCD has non-unit common factors with both cofactors. Wang's EEZGCD alleviates this problem by dividing out one common factor. Maple's implementation uses a trial-and-error strategy due to Spear [5, 8].

Zippel's Sparse Modular Interpolation Algorithm: This algorithm constructs GCDs by interpolating the coefficients. A "skeleton" of the GCD in the main variable is computed by a dense interpolation using the Chinese Remainder Algorithm. The correct coefficients are then computed by a fast sparse (Vandermonde) interpolation. Each of these integer coefficients is then recursively dense and sparse interpolated into polynomials involving the second variable and so forth. The sparse interpolation ignores the zero coefficients from the "skeleton" produced by the dense interpolation, and hence as originally formulated, the algorithm is fast but also probabilistic. This algorithm is not efficient for dense GCDs and could also be unlucky [10].

3 The Heuristic GCD Algorithm

Because it has been neglected to date, with the exception of the implementation in Maple, We describe the focus of the paper, GCDHEU, in greater detail than the other methods. Our implementation in Lisp is a direct translation from Maple's **gcd/gcdheu** procedure. The main concept is to reduce the number of variables by assigning values to each. The end result is a pair of fairly large integers. A guess at the polynomial GCD is then constructed one variable at a time using m-adic interpolations starting from the GCD of these two numbers. The following algorithm uses this heuristic and, returns the GCD of two polynomials and its cofactors or signals FAILURE.

PROCEDURE PGCDHEU (p,q)
Set $v_s \leftarrow \{\text{vars of } p\} \cap \{\text{vars of } q\}$
Set $g \leftarrow$ GCD(integer content of p, integer content of q)
WHEN v_s is empty,
  RETURN $\{g,\ p/g,\ q/g\}$
Set $p \leftarrow p/g$; Set $q \leftarrow q/g$
Set $v_1 \leftarrow$ first element of v_s
Set $\beta \leftarrow 2 \cdot \min(\| p \|_\infty, \| q \|_\infty) + 29$
Set $\xi \leftarrow \max(\min(\beta, 10000 \cdot \sqrt{\beta}/101),$
    $2 \cdot \min(\| p \|_\infty /|\mathrm{lc}(p)|, \| q \|_\infty /|\mathrm{lc}(q)|) + 2)$
LOOP
  Set $b_1 \leftarrow$ length(ξ)$\cdot$ max(degree(p,v_1), degree(q,v_1))
  WHEN $b_1 > 4000$
    signal FAILURE
  WHEN $b_1 > 400$ and length(v_s) > 1
    Set $m \leftarrow \max_{v \in v_s - \{v_1\}} \max(\text{degree}(p,v), \text{degree}(q,v))$
    Set $b_2 \leftarrow$ length(ξ) $\cdot m \cdot$
      (min(degree(p,v_1),degree(q,v_1)) + 1)

```
        WHEN b_2 > 8000
            signal FAILURE
    Set {γ, cofactorp, cofactorq} ←
        PGCDHEU(subs(v_1 ← ξ, p), subs(v_1 ← ξ, q))
    Set γ ← SP-interpolate(γ, v_1, ξ)
    Set γ ← γ/(integer-content of γ)
    WHEN γ divides both p and q
        RETURN {g · γ, p/γ, q/γ}
    Set cofactorp ← SP-interpolate(cofactorp, v_1, ξ)
    WHEN cofactorp divides p
        Set γ ← p/cofactorp
        WHEN γ divides q
            RETURN {g · γ, cofactorp, q/γ}
    Set cofactorq ← SP-interpolate(cofactorq, v_1, ξ)
    WHEN cofactorq divides q
        Set γ ← q/cofactorq
        WHEN γ divides p
            RETURN {g · γ, p/γ, cofactorq}
    Increment *GCD-TRIES*
    WHEN *GCD-TRIES* > 6
        signal FAILURE
    Set ξ ← 73794 · ξ · sqrt(sqrt(ξ))/27011
END LOOP
END PGCDHEU
```

```
PROCEDURE SP-interpolate(γ, v_1, ξ)
Set e ← γ
Set G ← 0
Set i ← 0
WHILE e ≠ 0 DO
        Set c_i ← e mod ξ
        Set G ← G + c_i v_1^i
        Set e ← (e − c_i)/ξ
        Set i ← i + 1
END WHILE
RETURN G
END SP-interpolate
```

4 GCD in Computer Algebra Systems

We concentrate on three prominent computer algebra systems' GCDs: Macsyma, Maple and Mathematica. We invite comparisons to other systems (for example, Axiom, Macaulay, Magma, Pari, Reduce, Senac, SAC-2). For our experiments here we used as a basis for comparison, our own Common-Lisp system "Mock-MMA". The Subresultant PRS and GCDHEU are implemented and used separately, and at the time of this writing, are quite straightforward implementations, not including any of the special-case optimizations we mention here.

Macsyma (version 419.0 from Macsyma Inc.) implements the following GCD algorithms: EZGCD, Subresultant PRS, Reduced PRS and SPMOD algorithms. The default is SPMOD. The user can choose another algorithm by setting a flag.

Maple's main GCD procedure employs these three algorithms: GCDHEU, GCDCHREM, and EZGCD with Spear's trial-and-error patch. The Maple program first uses GCDHEU. In the case of failure, Maple dispatches one and two-variable cases to the modular algorithm and the rest to the EZGCD algorithm. The recursive calls in EZGCD are to GCD. This means GCDHEU and GCDCHREM are used for smaller subproblems in EZGCD. In addition to this program, Maple also supplies an implementation of the Extended Euclidean algorithm, separate from the main GCD procedure.

Through private communications with Wolfram Research Inc., we have learned that Mathematica implements SPMOD. In the unlikely cases where SPMOD fails, Mathematica switches to the Subresultant PRS algorithm.

5 More Heuristics

Our own (RJF's) first experience with the modular GCD in Macsyma was in implementing the algorithm as described by W.S. Brown in a pre-print of his 1970 SIGSAM paper. In the summer of 1970 when it was first installed, most programs using GCDs ran slower. While we were slow to realize the cause, assuming some bug or "inefficient programming practice" to be the culprit, it turned out that for sparse multivariate polynomials, faster results were obtained by using the (then current) reduced PRS. It was becoming clear that a more devious approach to GCD computation would be advantageous, so long as the "customer base" was not randomly-generated dense multivariate polynomials.

In fact, we looked back at some literature on ALPAK, a precursor to ALTRAN designed by Brown and others, and discovered a number of interesting observations. The net result was the inclusion of heuristics which we believe made short work of most of the GCD computations by reducing large cases to small ones. Most of these tricks do not appear in the literature, although the major idea — keeping expressions in factored form as much as possible — has gained general acceptance as an important contributor to efficiency.

The current versions of Macsyma implement the following heuristics. Maple uses (at least) the first three) We suspect that Mathematica, too, uses some of them as well. them.

1. Reduce the degrees of homogeneous input polynomials. Notice that $p(x) = q(x^n)$ for $n > 1$, and rename x^n to y. This heuristic obviously generalizes to the multivariate case.

2. Take advantage of polynomials that have immediately obvious trivial factors (e.g. x^n or $x^n y^m$).

3. If the variable x occurs in polynomial p but not in q, then we can make x the main variable of p. Thus, the GCD of p and q is the GCD of q and the content of p. That is, x is eliminated from the computation.

4. Sort the variables according to their degrees. The lowest degree variable should be chosen as the main variable in the PRS computations. Maple's EZGCD implementation also uses other variable re-ordering heuristics such as designating as the main variable the variable with the "simplest" leading coefficient.

5. Assume $p = a \cdot x + b$ (where a and b are polynomials in other variables), and q is at least linear in x. We notice that the GCD is either linear or constant in x. The first step is to compute and divide out the contents of p and q (using x as the main variable). If the primitive part of p divides that of q, then the GCD is linear, hence it is the primitive part of p times the GCD of the contents. Otherwise, the GCD of p and q is just the GCD of the contents. This hack is structured more efficiently in Macsyma. For example, the content of q is never computed if $\mathrm{GCD}(a, b) = 1$.

Using these heuristic methods and hacks can be fruitless, but since this wasted time is generally only a small percentage of the time otherwise taken on hard problems, this is not something very worrisome.

We mention in passing that timing the routines in systems whose interior is concealed can be difficult. An incautious experimenter may end up testing only heuristics and not algorithms, (or even, in repeating tests for the sake of more accurate timing, encounter a system's caching of answers!).

6 Interweaving GCDs

It is clear that each of the GCD algorithms and heuristics or hacks has its own strengths and weaknesses. For example, SPMOD is much faster than EZGCD and GCDCHREM for very sparse GCDs. Given these different algorithms and heuristics, a present-day "best algorithm" would be a dispatching procedure that picks the correct methods according to its inputs. Maple automates this process to some extent. K. Geddes estimates that in the Maple test-suite, GCDHEU solves 90% of the inputs to GCD. However, GCDHEU is not the fastest in all these situations. An implementation of SPMOD is also likely to improve Maple's performance in sparse cases. Thus, a decision procedure more complicated than the one used by Maple is needed for optimal performance. This procedure could be called by the user, or during recursive calls to GCD in any of the algorithms, for example, when computing the contents.

What are the criteria for choosing an algorithm? We have not yet solved that problem. However, as discussed in the previous section, some quick hacks as a preliminary to any full algorithm seem advisable: Their failures only cost a small percentage of the time required to execute a modular algorithm.

Below is yet another set of heuristics that might be helpful in choosing the correct method. Some of these might be common sense. But we have not actually tested them.

1. If the common GCD of more than two polynomials is needed, avoid using PRS algorithms.

2. From empirical data, we might be able to determine the right algorithm for univariate cases based on the degree.

3. It should be reasonable to assume the GCD to be sparse. If one of the intermediate skeletons in SPMOD is dense, then switch to another algorithm.

4. If it is known in advance that the GCD has small coefficients, then pick small evaluation points for GCDHEU. If bignums are avoided altogether, GCDHEU can be much faster. An older version of our GCDHEU implementation was mistakenly compiled to assume fixnum coefficients. The program was unusually fast, and produced correct answers for our test cases. However, it has been pointed out that wrongfully assuming small coefficient can result in incorrect answers from GCDHEU [5].

7 Timing and Discussion

We have taken the seven cases from Yun's Ph. D. thesis [9] on EZGCD for our experiments. In all these cases, the GCD is the polynomial D, and the inputs are F and G.

In the tables below, the times shown are in milliseconds. The first row in each table, labeled **pgcdsr**, shows the running times of our Mock-MMA Subresultant PRS program. The second row, labeled **pgcdheu** shows the running times of our GCDHEU procedure. In the third row, labeled **Maple**, are the running times of the Maple GCD procedure. The fourth rows, labeled **Macsyma**, are the running times of Macsyma's SPMOD GCD procedure. The fifth row, labeled **Macsyma:prs**, are the running times of Macsyma's Subresultant PRS implementation. The last row, labeled **Math** show the running times of Mathematica's `PolynomialGCD` operation.

The blank entries in the **pgcdheu** row denote cases where our program signalled *failure*, usually rather promptly, except for $v = 10$ in case 5 where the computer ran out of memory. The blanks in the other rows were indicate the running times were far longer other algorithms in the same column.

We also attempted to time Macsyma's EZGCD procedure. But it seemed to run forever for cases 1, 5, 5' and the smallest example in case 4. Consequently, we have decided against using the data produced from executing this procedure.

Macsyma's EZGCD was unfortunately, the only "purely EZ" implementation known to us. Maple's GCD procedure employs the EZGCD method only when all its heuristics fail. From our data, fewer than half of the examples suggest the use of EZGCD. To draw conclusions regarding EZGCD based on these numbers cannot be totally justified, because Maple's EZGCD uses the heuristics recursively.

Mock-MMA is compiled in Allegro Common Lisp. The commercial Macsyma 419.0 is compiled using Austin-Kyoto Common Lisp (AKCL). In both cases we include garbage-collection time when it occurs. We estimate this slightly understates the cost for the smaller problems (where GC does not occur), slightly overstates the time of some medium-sized problems, where GC occurs once, and is about right for large-sized problems where multiple GCs occur, almost all of which were required by that problem.

Getting consistent repeatable timing for Maple is also tricky. Maple stores results from various subroutines in "remember tables." In most cases, these are cleared at the occasional garbage collection. Hence the remembered results will disappear at unpredictable times. Re-invocation of a subroutine with the same input data as a previous invocation will immediately return the remembered result if it has not been garbage-collected. Thus in Maple, an incautious experimenter may encounter inconsistent timing results.

It should also be noted that Maple sometimes has to load a library procedure at its first use, and the time to do this is credited to that procedure invocation. For example, the procedure **gcd/gcdeh** is loaded into memory at its first invocation. From our experiments, we have found no significant increases in running times due to the loading of procedures.

All programs were run on HP 9000 workstations. Specifically, Macsyma was run on a HP 9000/715 workstation, Mathematica on an HP 9000/755, Maple on an HP 9000/712. Our own Mock-MMA procedures were timed on all three machines. Since these HP machines executed the same binary programs, their speed ratios were close to constant. The model 715 and model 755 were each 1.8 times faster than the model 712. The running times in the following tables have already been adjusted to reflect this speed ratio and correspond to times on the model 712.

Case 1: GCD = 1.

$$F = (x + \sum_{i=1}^{v} y_i + 1)(x + \sum_{i=1}^{v} y_i + 2)$$

$$G = (x^2 + \sum_{i=1}^{v} y_i^2 + 1)(-3y_1 x^2 + y_1^2 - 1)$$

v	1	2	3	4	5
pgcdsr	6	**8**	**15**	**27**	**48**
pgcdheu	**4**	**8**	17	32	63
Maple	50	60	60	80	160
Macsyma	90	108	504	540	558
Macsyma:prs	36	36	54	72	90
Math	18	36	54	90	144
v	6	7	8	9	10
pgcdsr	95	**144**	**221**	**330**	**464**
pgcdheu	173	439	1352	4701	
Maple	250	430	860	2480	10180
Macsyma	630	1062	1134	1602	1746
Macsyma:prs	**90**	180	612	702	774
Math	234	432	792	1386	2628

Note that for this and subsequent tables, the fastest time is indicated in **boldface**. In this case we expected all modular algorithms, EZGCD, SPMOD and GCDHEU should do well since the GCD is 1; the chance of an unlucky homomorphism is practically zero. They were, however, rather slow. Pgcdheu and hence Maple should be slow as v increases. Our Subresultant PRS procedure and Macsyma's PRS appear to have the edge.

Note when $v = 10$, our **pgcdheu** failed while Maple's GCDHEU succeeded. This discrepancy is due to the difference in variable orderings in the two implementations, since this example is not symmetric in the variables.

Case 2: Linearly dense quartic inputs with quadratic GCDs.

$$D = (x + \sum_{i=1}^{v} y_i + 1)^2$$

$$F = D \cdot (x - \sum_{i=1}^{v} y_i - 2)^2$$

$$G = D \cdot (x + \sum_{i=1}^{v} y_i + 2)^2$$

v	1	2	3	4	5
pgcdsr	**12**	43	135	296	**581**
pgcdheu	14	51	311	3316	
Maple	60	150	600	2780	2980
Macsyma	126	756	2916	7596	19620
Macsyma:prs	18	**36**	**108**	**162**	684
Math	36	162	360	882	1908
v	6	7	8	9	10
pgcdsr	1054	1829	2973	4630	6977
pgcdheu					
Maple	4900	7580	12950	22910	34510
Macsyma	44928				
Macsyma:prs	**846**	**1530**	**2736**	**3546**	**3996**
Math	4086	8244	15732	33228	62766

Maple switched from GCDHEU to EZGCD starting at $v = 5$.

We observed that GCDHEU reached its bounds very quickly, and did not perform very well on the cases where it succeeded.

Due to the density in the GCD, EZGCD is faster than SPMOD in this case. We see that the running times for Mathematica's SPMOD implementation is approximately proportional to powers of 2, while EZGCD is approximately proportional to powers of 1.5.

Macsyma's Subresultant PRS was the fastest in this case. Our implementation of the same algorithm was a close second. The PRS required only two pseudo-divisions in the main variable, regardless of the value of v.

Case 3: Sparse GCD and inputs where degrees are proportional to the number of variables.

$$D = x^{v+1} + \sum_{i=1}^{v} y_i^{v+1} + 1$$

$$F = D \cdot (x^{v+1} + \sum_{i=1}^{v} y_i^{v+1} - 2)$$

$$G = D \cdot (x^{v+1} + \sum_{i=1}^{v} y_i^{v+1} + 2)$$

v	1	2	3	4	5
pgcdsr	**4**	**8**	**18**	**32**	**54**
pgcdheu	7	48			
Maple	50	100	130	210	260
Macsyma	54	666	1926	4554	10692
Macsyma:prs	18	18	36	36	**54**
Math	18	36	72	90	144
v	6	7	8	9	10
pgcdsr	103	148	212	301	412
pgcdheu					
Maple	400	830	1300	3360	9310
Macsyma	20970	37440	66438		
Macsyma:prs	**36**	**72**	**72**	**108**	**126**
Math	216	342	504	792	1278

The sparsity should make SPMOD a good method in this case, as observed for Mathematica's implementation. We do not have a good explanation for Macsyma's sluggishness here.

Because the inputs are of high degrees, our implementation of GCDHEU failed starting at $v = 3$. Maple's GCDHEU, on the other hand, finished all ten cases. It benefited from substituting $x \leftarrow x^{v+1}$ and $y_i \leftarrow y_i^{v+1}$, resulting in linear sparse polynomials. The sparsity helps GCDHEU because its performance depends heavily on values of the GCDs at large evaluation points.

Our data concurs with Yun's data showing that PRS is very efficient on this case as well.

Case 3': Same as 3, except

$$G = D \cdot (x^v + \sum_{i=1}^{v} y_i^v + 2)$$

v	1	2	3	4	5	6	7	8	9	10
pgcdsr	8	119	83520							
pgcdheu	**7**	**40**								
Maple	50	130	480	680	630	1010	**1060**	**1460**	**1580**	**2260**
Macsyma	54	666	1998	6048	12114	25272	47052	80748		
Macsyma:prs	36	108	3258	119574						
Math	36	72	**126**	**252**	**450**	**828**	1422	2376	3906	6282

In this variation, Yun's data suggested major performance degradations in the PRS algorithm, and minor slowdowns in the EZGCD algorithm, compared to Case 3. Maple switched to EZGCD starting at $v = 3$. Indeed, the slowdowns are minor.

The heuristic GCD, again, suffers from high degree inputs. Maple was not able to use its substitution heuristic in this case, and hence, switched to EZGCD.

This case shows that for certain kinds of large multivariate problems the modular algorithms are indeed necessary. We do not know why SPMOD slowed down in Macsyma, given such sparse GCDs. Mathematica's implementation of SPMOD, on the hand, performed only second to Maple's EZGCD.

Case 4: Quadratic non-monic GCD. F and G have other quadratic factors.

$$D = y_1^2 x^2 + \sum_{i=2}^{v} y_i^2 + 1$$

$$F = D \cdot (x^2 - y_1^2 + \sum_{i=2}^{v} y_i^2 - 1)$$

$$G = D \cdot (y_1 x + \sum_{i=2}^{v} y_i + 2)^2$$

v	1	2	3	4	5	6	7	8	9	10
pgcdsr	18	119	848	5709	26381					
pgcdheu	**12**	**29**	**106**	684						
Maple	50	100	280	2300	3560	2740	3220	3980	**4720**	**6100**
Macsyma	162	1116	1332	2664	5616	9126	16596	27504	41904	65790
Macsyma:prs	36	90	1134	2736	16452	73854	411534			
Math	36	72	162	**306**	**612**	**1134**	**2142**	**3762**	6480	11286

Maple switched from GCDHEU to EZGCD starting at $v = 5$.

This is the case where EZGCD should have trouble with the non-monic GCD. But for large cases ($v \geq 9$), EZGCD was surprisingly the fastest. It is interesting to see Maple speeding up from $v = 5$ to $v = 6$. Perhaps it fiddled too long with the heuristics, but gave up faster for $v \geq 6$ and used EZGCD.

We see that GCDHEU can be useful for small values of v.

Case 5: Completely dense non-monic quadratic inputs with dense non-monic linear GCDs.

$$D = (x+1)\prod_{i=1}^{v}(y_i + 1) - 3$$

$$F = D \cdot ((x-2)\prod_{i=1}^{v}(y_i - 2) + 3)$$

$$G = D \cdot ((x+2)\prod_{i=1}^{v}(y_i + 2) - 3)$$

v	1	2	3	4	5	6	7	8	9	10
pgcdsr	10	190	4442	203060						
pgcdheu	**9**	**31**	**123**	**427**	**1527**	**5853**	**22983**	**95480**	**411010**	
Maple	60	310	380	1130	4380	17530	77260	398260		
Macsyma	576	972	4554	16776	73008	375246				
Macsyma:prs	36	216	4968	165510						
Math	36	126	630	4266	38178	398682				

Obviously, GCDHEU outperformed the others in these cases. Both pgcdheu's and Maple's running times roughly quadruple as v increases. Our compiled version was 3 to 4 times faster than Maple's.

Maple handled all eight (8) examples with GCDHEU. As an experiment, we forced Maple to use EZGCD for $v = 8$. On our HP workstation, Maple computed an obviously incorrect answer (amazingly) in 760 milliseconds. Furthermore, the answer in this case was presented in such a form that computing the difference between it and D, the GCD, ran out of memory. However, the same experiment on an DEC Alpha showed that GCDHEU took 103 seconds while EZGCD took 540 seconds. Thus, the decision by Maple to stick with GCDHEU was the correct one.

The modular algorithms were starting to outperform the PRS at $v = 2$. But beyond $v = 6$, we ran out of patience. Due to the dense GCD, SPMOD was expected to be slow. Its running time seemed to be approximately proportional to powers of 7 for Macsyma and powers of 10 for Mathematica.

Case 5': Sparse non-monic quadratic inputs with linear GCD's.

$$D = x\prod_{i=1}^{v} y_i - 1$$

$$F = D \cdot (x\prod_{i=1}^{v} y_i + 3)$$

$$G = D \cdot (x\prod_{i=1}^{v} y_i - 3)$$

v	1	2	3	4	5
pgcdsr	**3**	**5**	**9**	**13**	**17**
pgcdheu	6	14	27	47	81
Maple	50	60	100	110	200
Macsyma	90	180	684	1206	1350
Macsyma:prs	18	18	18	36	18
Math	18	18	36	54	90
v	6	7	8	9	10
pgcdsr	23	32	38	48	90
pgcdheu	165	420	1378	5194	20497
Maple	310	700	2350	7260	28010
Macsyma	1908	2502	3096	3726	4320
Macsyma:prs	**18**	**18**	**36**	**18**	**36**
Math	162	324	540	1098	2124

In this case, our **pgcdsr** was fastest for the smaller examples ($v \leq 5$), and Macsyma's implementation was fastest for larger examples. We speculate that Macsyma advantage was in using a polynomial representation more efficient for sparse polynomials.

Heuristic GCD did not seem to have much advantage over the modular algorithms. But our GCDHEU for this case, is about 50% to 400% faster than Maple's. (Maple never switched to EZGCD.)

Case 6: Trivariate inputs with increasing degrees.

$$D = x^j y(z-1)$$
$$F = D \cdot (x^j + y^{j+1} z^j + 1)$$
$$G = D \cdot (x^{j+1} + y^j z^{j+1} - 7)$$

j	1	2	3	4	5
pgcdsr	24	49	123	514	2441
pgcdheu	**18**	25	**35**	54	95
Maple	80	100	100	80	80
Macsyma	**18**	72	72	72	72
Macsyma:prs	**18**	**18**	36	**18**	**36**
Math	36	36	36	36	**36**
j	6	7	8	9	10
pgcdsr	10976	42480	146160		
pgcdheu	170	296	528	916	1497
Maple	130	150	210	300	410
Macsyma	54	90	72	72	72
Macsyma:prs	**36**	**36**	54	54	**36**
Math	54	54	**36**	**36**	36

In this case, the GCD has only two terms. Thus the sparse algorithm should be much faster than the PRS. But the Maple, Macsyma and Mathematica all took advantage of the obvious factor $x^j y$, and trivialized this case. Thus, they all outperformed the Mock-MMA programs. In particular, a direct comparison between our **pgcdsr** and Macsyma's PRS program shows the advantage of this heuristic.

After finding the trivial factor, Maple solved all ten examples using GCDHEU. But it is interesting to note that our GCDHEU, not having found the trivial factor, was still competitive with Maple's version for $v \leq 4$. This suggests compilation can improve the performance of GCDHEU.

Case 7: Trivariate polynomials whose GCD has common factors with its cofactors.

$$P = x - yz + 1$$
$$Q = x - y + 3z$$
$$D = P^j Q^j$$
$$F = P^j Q^k$$
$$G = P^k Q^j, \text{ where } j < k.$$

(j,k)	(1,2)	(1,3)	(1,4)
pgcdsr	42	722	17090
pgcdheu	**26**	**43**	**77**
Maple	100	150	230
Macsyma	684	882	2034
Macsyma:prs	54	324	18630
Math	72	144	234
(j,k)	(2,4)	(3,4)	(5,10)
pgcdsr	1875	363	
pgcdheu	**144**	265	
Maple	460	1030	94280
Macsyma	2214	2196	103518
Macsyma:prs	738	**234**	
Math	612	1440	**77256**

This case was originally designed to test the special patch of EZGCD where the GCD and both of its cofactors are not relatively prime. The only occurrence of this problem in Maple was when $j = 5$ and $k = 10$. This is also the only case where Maple's GCDHEU failed.

The Heuristic GCD was fastest for small cases. And our compiled version outperformed Maple's interpreted version by a factor between 3 and 4.

8 Conclusion

From the data and the discussion in the previous section, we draw five simple conclusions. We are confident that the running times we observed, in spite of timing uncertainties, had less than 30% error, while we noted performance differences much greater than that.

- **Heuristic GCD is useful and should be considered for implementation in all computer algebra systems.** It is clear from our data that our implementation of GCDHEU was the clear winner in case 5. It was the fastest for small examples in cases 4 and 7. And it was faster than Mathematica and Macsyma's SPMOD in case 1 for $v \leq 6$ and in case 5' for $v \leq 5$. Maple's partially-interpreted GCDHEU also outperformed Macsyma and Mathematica's SPMOD implementations in case 5.
- **Good compilation is necessary.** Our data suggests that our compiled GCDHEU can be up to 4 times faster than Maple's interpreted version. Not counting cases 3 and 6 where Maple used other heuristics, our implementation outperformed Maple's in 39 out of 44 examples. (This comparison should be taken cautiously, since the two programs represent polynomials differently and may be sensitive to the ordering of variables.)
- **Other heuristics improve performance.** We see from case 6 that the commercial programs were very fast in removing obvious factors, thus speeding up their GCD computations, resulting in much faster execution than our procedures. We also see from case 3 that obvious substitutions also improve performance dramatically.
- **The Subresultant PRS algorithm is still useful.** Despite the theoretical advantages of modular algorithms, the two implementations of the Subresultant

PRS algorithm in Mock-MMA and Macsyma were consistently the fastest for 4 out of 9 cases, namely cases 1, 2, 3 and 5'. It is known that both Maple and Mathematica have implementations of this algorithm, but they are rarely used.

- **Dense representation can be advantageous for small problems.** We believe that our Subresultant PRS program (pgcdsr) outperforms Macsyma's version for small cases because our dense representation, using recursive arrays, is more suitable for the pattern of memory accesses best supported in modern computer architectures. On the other hand, for large sparse problems, the simplicity of Macsyma's recursive linked lists appears superior. This suggests that there are benefits for dense and sparse representations coexisting in computer algebra systems.

We are unable to propose a neat decision procedure for choosing the optimal GCD algorithm to use. But it is clear that heuristics and hacks are very likely to solve very small problems much faster than complicated algorithms. Heuristics too, can employ other heuristics. As mentioned earlier, a bug we initially encountered in our implementation of the GCDHEU procedure, where we unreasonably assumed the GCD had small coefficients, turned out to be a big timesaver in those instances in which it did not signal failure. Although it did not produce any errors for us, it could do so. Given its speed advantage though, it may pay to "open compile" a fixnum-specific version of GCDHEU to be used when possible.

All code used in Mock-MMA can is available from the authors.

9 Acknowledgments

We thank Daniel Lichtblau of Wolfram Research Inc. for his information on the GCD methods used in Mathematica. We also thank Keith Geddes for his encouragement and explanations of the fine details in Maple.

References

[1] W. S. Brown, "On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors," *J. ACM*, 18(4), pp. 476-504, 1971.

[2] Shang-Ching Chou, *Mechanical Geometry Theorem Proving*, D. Reidel Publishing Co., Dordrecht Holland, 1988.

[3] G. E. Collins, "Subresultants and Reduced Polynomial Remainder Sequences," *J. ACM*, 14(1), pp. 128-142, 1967.

[4] R. Fateman, "A Lisp-language Mathematica-to-Lisp Translator," *ACM SIGSAM Bulletin* 24, no. 2, pp. 19-21 (April, 1990). Also reprinted in *Computer Algebra Nederland Nieuwsbrief 6*, October, 1990. Code availably by anonymous ftp.

[5] K.O. Geddes, S.R. Czapor, and G. Labahn, *Algorithms for Computer Algebra*, Kluwer Academic Publishers, Boston 1992.

[6] D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms* (second edition), Addison-Wesley, Reading 1981.

[7] Guy L. Steele Jr. *Common Lisp the Language* (second edition), Digital Press, 1990.

[8] P. S. Wang, "The EEZ-GCD Algorithm," *ACM SIGSAM Bulletin* 14, pp. 50-60, 1980.

[9] David Y. Y. Yun, *The Hensel Lemma in Algebraic Manipulation*, MIT Technical Report MAC TR-138, Project MAC, MIT 1973.

[10] Richard Zippel, *Effective Polynomial Computation*, Kluwer Academic Publishers, Boston 1993.