NON-MODULAR COMPUTATION OF POLYNOMIAL GCDS USING TRIAL DIVISION

Anthony C. Hearn
Department of Computer Science
University of Utah
Salt Lake City, Utah U.S.A.

1. INTRODUCTION

This paper describes a new algorithm for the determination of the GCD of two multivariate polynomials by non-modular means. This algorithm, which is now available in the current distributed version of REDUCE, is believed to be superior to all other published non-modular algorithms, including the improved sub-resultant algorithm [1]. It also competes effectively with the EZ [2] and modular GCD algorithms [3] for many applications. Moreover, it has the advantage over these latter algorithms in that it may be specified in a very compact form, and thus is suitable for applications which have constraints on the amount of code used.

The algorithm arose from an empirical study of the behavior of the improved sub-resultant algorithm which has been used in REDUCE for several years now. The current state of this and other non-modular algorithms is described in detail in two excellent papers by Brown [1,3] and for simplicity in exposition, I shall assume his notation. Brown points out that all non-modular algorithms arise from adjustments to α_i and β_i in the definition of a polynomial remainder sequence (prs) F_i defined via the equation

$$\beta_i F_i = \alpha_i F_{i-2} - Q_i F_{i-1} \quad , \tag{1}$$

where we assume that F_1 and F_2 are primitive. In the usual treatment, the Q_i are obtained by pseudo-division and hence the α_i are given by

$$\alpha_i = f_{i-1}^{\delta_{i-2}+1} \tag{2}$$

where f_i is the leading coefficient of the main variable of F_i, $\delta_i = n_i - n_{i+1}$, and n_i is the degree of the main variable in F_i. The problem then is to choose the largest β_i so that F_i is still a polynomial. Clearly the best choice is

$$\beta_i = \text{cont}(\text{prem}(F_{i-2}, F_{i-1})) \qquad (3)$$

but this requires the calculation at each step of the GCD of all coefficients of the left hand side of Eq. (1) which is usually assumed to be too expensive. We shall, however, return to this point later. The two most popular such algorithms are the reduced prs of Collins [4] which chooses

$$\beta_i = \alpha_{i-1} \qquad (4)$$

and the sub-resultant algorithm of Collins and Brown, which has a much more complicated formula for β_i than Eq. (4) but which is shown in [3] to be more efficient than the reduced prs in the case when the prs was abnormal, i.e, $\delta_i > 1$.

The standard derivation of the prs iterates over degrees of the relevant polynomials; however, as I pointed out in a previous paper [5] a simpler prs results in the abnormal case if one iterates over the non-zero terms in the polynomial rather than indexing over degrees. Consequently, if any powers of the main variable are missing during the pseudo-division iterations, either by chance cancellation of coefficients or the absence of powers in the original polynomials, one finishes with fewer powers of f_i in the coefficient α_i in Eq. (1), and therefore the remainder term $\beta_i F_i$ is smaller.

The problem now however is that the original derivation of β_i for both the reduced and the sub-resultant formulas is no longer valid, since these depended on the explicit form of α_i given in Eq. (2). Hence in either case a new derivation is necessary.

In my previous paper, I pointed out that the basis for such a derivation in the sub-resultant case was already given in Brown's 1971 paper [3], and using this, a program for GCD computation using term iteration was presented. However, the derivation of the formula in this program was incomplete and in certain rare cases the program would fail by computing too large a β_i. In fact, the complete solution as given recently by Brown [1] requires rational factors for β_i ! Since this formulation involves a rather messy computation to determine β_i, I decided to look again for a simpler method of calculating GCDs.

2. THE DERIVATION OF A NEW ALGORITHM

For several years now, I have studied the behavior of a large number of examples of GCD computations to see if any empirical evidence

could be accumulated which might result in a simpler and hopefully better algorithm. As a result of these studies, the following conclusions have been reached:

1) The dominant cost in non-modular GCD computations is in the processing of the coefficients in the members of the prs. The reason for this is that the pseudo-division step in the remainder calculation requires the formation of the product of such coefficients, which has the potential for explosive expression growth. In the worst case, this growth is "severely superexponential" [1]. As a result, anything done to keep such coefficients small is worth the effort.

2) Iteration over terms rather than degrees helps in this regard since in general the expressions processed in the pseudo-division step are significantly smaller in the abnormal case.

3) For multivariate polynomials, such coefficients are also smaller when the main variable is that of lowest degree in the two polynomials. This observation is already well known, and is intuitively obvious since we want to keep the series in the largest number of variables as short as possible. More specifically, we need to minimize the difference between the leading degree and the trailing degree (ie, the minimum degree of the variable in the polynomial). In addition, variables present in one polynomial but not the other should be given highest weight in the polynomial in which they occur, since the algorithm will then exclude such variables in the most efficient manner.

4) The introduction of products of powers of leading coefficients in the pseudo-division step (the f_i in Eq. (2)), manifests itself by the appearance of these factors in later terms in the prs. This is of course the basis of both the reduced and sub-resultant algorithms, and again is well known.

5) The only factors which the existing non-modular algorithms find are a subset of the products of powers of leading coefficients introduced by the pseudo-division process. The discussions in the literature then have centered on ways to calculate the optimal β_i from a combination of these coefficients. It is worth noting, however, that since the improved sub-resultant prs factor actually involves ratios of powers of leading coefficients, it is possible that a quotient of two such coefficients is a multiplicative factor in β_i. However, it has been my experience that when the algorithm introduced in this paper is applied to a GCD computation, no additional factors arising from such ratios have been valid divisors. This suggests that the ratio of coefficients

needed in the sub-resultant algorithm only compensates for its failure to remove factors from earlier terms; this however is only a conjecture.

6) The word "abnormal" in describing a prs in which some terms are missing is a scientific misnomer although mathematically sound; in most practical calculations, this is the more common case (ie, the "normal" case) because of the structure introduced into the calculation by the various physical laws in nature.

With these principles in mind, one can now show the top level structure of an optimal non-modular algorithm for computing the GCD of two multivariate polynomials. However, before we do this we must first specify the data-structures used to define a multivariate polynomial. We shall use for this purpose the standard REDUCE polynomial representation in which a polynomial is either a constant or has the form:

$$u(x) = \sum_{i=0}^{n} u_i x^i = u_n x^n + \sum_{i=0}^{n-1} u_i x^i \qquad (5)$$

We refer to the first term in Eq (5) as the leading term, (or lt), and the second as the reductum (or red). The components of the first term are u_n, the leading coefficient (or lc) and x^n, the leading power (or lpow) whose components in turn are x, the main variable (or mvar) and n, the leading degree (or ldeg). In REDUCE, the "variables" in such forms can be quite general [6] and are referred to as kernels. A constant polynomial is discriminated by a boolean function domainp. Finally, the leading coefficient is either a constant or a non-constant polynomial in variables other than x . This description is sufficient for our purposes; readers who require more details are referred to [6].

In terms of such a polynomial structure our GCD algorithm now takes the form:

```
polynomial procedure gcd(u,v);
   begin dcl x:list of kernel;
      if domainp(u) then return gcdd(u,v)
       else if domainp(v) then return gcdd(v,u);
      x := setorder(leastkernelorder(u,v));
      u := gcdl(reorder(u),reorder(v));
      setorder(x);
```

```
    return reorder(u)
end;
```

The above definition is written in a documentation form of Mode Reduce [7] which we have found to be a convenient language for the specification of algorithms as well as being usable as an implementation language. The combining forms and declarations used above should be self-explanatory once one knows that the procedure definition declares its arguments and value as polynomial by default. For further details, readers are referred to [7] and [8].

In the above definition, we assume that leastkernelorder returns a list of kernels ordered such that one member precedes another if and only if it occurs in both u and v with a smaller degree. Setorder takes such a list and globally sets this as the kernel order in all subsequent operations on polynomials, with the first member of the list the main variable. It returns the previous kernel order. Gcdd takes a domain element and a polynomial as arguments and returns their GCD. It may be defined as follows:

```
polynomial procedure gcdd(u,v);
   %u is a domain element;
   if domainp(v) then gcddd(u,v)
    else gcdl(red(v),gcdd(u,lc(v));
```

Finally, we assume that the function gcddd which takes two domain elements and returns their GCD, is provided in the existing polynomial package.

The procedure gcdl above is the main recursive driving routine for the algorithm. It may be defined as follows:

```
mode content = struct(head:polynomial,tail:polynomial);

polynomial procedure gcdl(u,v);
   if domainp(u) then gcdd(u,v)
    else if domainp(v) then gcdd(v,u)
    else if divchk(u,v) ≠ 0 then v
    else if divchk(v,u) ≠ 0 then u
    else if mvar(u)=mvar(v)
     then begin dcl x,y:content;
     x := comfac(u);
     y := comfac(v);
     u := gcdl(tail(x),tail(y))
              *primgcd(u div tail(x),v div tail(y));
```

```
        if head(x) ≠ 1 and head(y) ≠ 1
          then if ldeg(head(x))>ldeg(head(y))
                 then u := head(y)*u
                 else u := head(x)*u;
           return u
        end
      else if mvar(u)>mvar(v) then gcdl(tail(comfac(u),v)
      else gcdl(u,tail(comfac(v));

content procedure comfac(u); dcl u:polynomial;
   begin dcl x:polynomial,y:kernel;
      if red(u)=0 then return (lpow(u) . lc(u));
      x := lc(u);
      y := mvar(u);
   A: u := red(u);
      if deg(u,y)=0 then return (1 . gcdl(x,u))
       else if red(u)=0 then return (lpow(u) . gcdl(x,lc(u)))
       else if x ≠ 1 then x := gcdl(lc(u),x);
      go to A
   end;
```

The procedure divchk used in the definition of gcdl returns u div v if v divides u or 0 otherwise. Its use is not really necessary and the two references may be omitted if desired. We shall return to this point later. Deg(u,var) is a procedure which returns the leading degree of var in u if var is the main variable of u or 0 otherwise. Its definition is as follows:

```
integer procedure deg(u,var);
   dcl u:polynomial,var:kernel;
   if domainp(u) or mvar(u) ≠ var then 0 else ldeg(u);
```

The only part remaining is the definition of the procedure primgcd for the computation of the GCD of two primitive non-constant polynomials with the same main variable. This is of course the crucial procedure for limiting coefficient growth, and so again in terms of the principles we have enunciated above we can now attempt to produce an optimal algorithm.

A possible choice consistent with our observations is the primitive remainder sequence algorithm (or prim). Although it is widely believed to be prohibitively expensive in general, we are now using it in a context where the coefficient sequence should be much smaller than in the more classical approach. The form for primgcd in this case is as follows:

```
polynomial procedure primgcd(u,v);
   begin dcl var:kernel,w:polynomial;
      if domainp(u) or deg(v,var := mvar(u))=0 then return 1
       else if ldeg(u)<ldeg(v) then {w := v; v := u; u := w};
  A:  w := remf(u,v);
      if w=0 then return v else if deg(w,var)=0 then return 1;
      u := v; v := pp(w);
      go to A
   end;

polynomial procedure pp(u);
   %returns the primitive part of non-constant polynomial u;
   u div tail(comfac(u));

polynomial procedure remf(u,v);
   begin dcl fl:polynomial,k,n:integer,var:kernel;
      fl := lc(v);
      var := mvar(u);
      n := ldeg(v);
      while (k := deg(u,var)-n) ≥ 0
       do u := fl*red(u)-var↑k*lc(u)*red(v);
      return u
   end;
```

In order to test the effectiveness of this algorithm, I took the set of six examples considered by Moses and Yun in assessing the EZ GCD algorithm [2] and used their numbers as a benchmark for testing various definitions for primgcd. The results are shown in the Appendix. We note that in the context of our driving routines, the prim algorithm is remarkably effective, except in the third example. I shall discuss this case later in Section 3. The reason of course that the prim algorithm works so well now is that we have kept the coefficient growth small by the choice of main variable and by iterating over terms. The prim coefficient is then reduced to the absolute minimal form for this class of algorithm and so we get the advantages of that in the succeeding prs member computations.

The question then is can one really do any better? Our only hope, again within this class of algorithms, is to reduce the size of the members of the prs by a relatively cheap computation before the final reduction to primitive form. The key to this lies in observation 5) above. After all, if all the previous non-modular GCD algorithms can do is determine a combination of products of powers of leading coefficients of previous terms in the prs, why not keep a list of such coefficients and use trial division to remove as many as possible? At

first sight, this might again seem too expensive, since trial division has the potential for intermediate expression swell in extreme cases, because we again compute a product of polynomials during the division process. Again in response we can only appeal to empirical evidence; we have not seen any cases in practice where the cost of any trial division we have used is a significant part of the GCD computation time. In fact, since one terminates the process as soon as a failure to divide is detected, the extra division often takes negligible time. This then is the reason for the two division checks in our top level procedure gcdl. It often happens that one of the two arguments of gcdl divides the other, and so it is worthwhile checking first to eliminate this case.

This then leads to the following new algorithm for GCD computation (referred to in our tables as the "basic" algorithm) in which only trial division is used to reduce the size of the prs coefficients.

```
polynomial procedure primgcd(u,v);
   %u and v are primitive polynomials, value is gcd(u,v);
   begin dcl var:kernel,w,x:polynomial,
         lclst:list of polynomial;
      if domainp(u) or deg(v,var := mvar(u))=0 then return 1
       else if divchk(u,v) ≠ 0 then return v
       else if divchk(v,u) ≠ 0 then return u
       else if ldeg(u)=1 or ldeg(v)=1 then return 1
     %since if the degree is 1, and u and v are primitive,
      the GCD must be 1 unless one polynomial divides the other;
       else if ldeg(u)<ldeg(v) then {w := v; v := u; u := w};
A:   w := remf(u,v);
      if w=0 then return pp(v) else if deg(w,var)=0 then return 1;
      if v ≠ 1 and v ≠ -1 then lclst := v . lclst;
      if (x := divchk(w,lc(w))) ≠ 0 then w := x
       else for each y in lclst
      do while (x := divchk(w,y)) ≠ 0 do w := x;
      u := v; v := w;
      go to A
   end;
```

It should be noted that we have also included a check for division by the leading coefficient of the newly constructed term in the prs. This happens sufficiently often that the additional overhead involved in making this check is well worth the effort.

The results for this algorithm are also shown in the Appendix. We see that in general this is not as good as the prim algorithm, and in some cases is decidedly worse. However, tests have shown that the additional cost comes from the failure of the trial divisions to recognize all factors in the coefficients, which only the primitive computation can find. The cost of the trial division computations themselves are not significant. This then suggests our third and final algorithm, in which we first use trial division to remove such factors and then apply the primitive reduction to what is left, which in general will be much smaller. The new definition of primgcd requires only two changes to the previous one, namely the line after label A is replaced by:

```
if w=0 then return v else if deg(w,var)=0 then return 1;
```

and the last line but two replaced by:

```
u := v;  v := pp(w);
```

The results of using this algorithm (referred to as "full" in the tables) are also shown in the Appendix. Overall, it performs better than the prim algorithm. Furthermore, in the cases in which it performs worse, the difference is small, giving us in fact a measure of the overhead introduced by the trial divisions. Our third form of primgcd above then is our algorithm of choice, although the simpler prim algorithm is certainly acceptable. However, we must emphasize again that these results are only valid if:
1) the main variable has the lowest degree of all variables
2) iteration is done over terms and not degrees.

3. COMPUTING TIME CONSIDERATIONS

The next question to resolve is why these algorithms do so badly in the third example in the Moses-Yun set. On inspection, this case turns out to be the very worst for our assumptions, namely:
1) All variables appear with the same degree
2) Until the very last step in the prs computation, the coefficients grow exponentially in size, but have no common factors.
Clearly, the non-modular approach as we have presented it cannot do well in such cases. So, any computing time analysis has to conclude with Brown [1] that in the worst case our algorithms are superexponential in nature. Fortunately, the worst case is not common in practice. In the best case, the algorithm is better than linear as seen from the examples. Brown considers several models of computation to analyze the sub-resultant algorithm and we could also do the same

here. However, we prefer to let our case rest on our empirical results than to try to parameterize all possible ways in which the algorithm can behave.

4. CONCLUSION

I think that one of the most exciting things about these results is that they show that the search for an "optimal" polynomial GCD algorithm is still not over. There will surely be improvements made to this algorithm as well as in the Hensel construction and modular methods. Since GCD computation is so fundamental in many algebraic computations, any improvements will make more calculations possible with a given amount of computing resources. However, one might question now whether the methods we have discussed here could be made significantly more efficient since the series we have produced is now minimal in size. In response, I believe that we can still go a long way further in the cases where significant expression swell occurs. The reason we have reached a limit now is that we have assumed implicitly in all steps of our computation that expanded polynomials were necessary. It was these expanded polynomials which gave us the coefficient growth exhibited, say, in the third benchmark. One question to ask is whether expanded polynomials are really necessary; in other words, can we formulate an algorithm in which a more structured form is used for the polynomial? I have already discussed this approach for the elementary operations on polynomials such as addition and multiplication [9] and shown that in a wide range of calculations the computing time and space requirements can be dramatically decreased. The next step clearly is to apply such methods to more complicated algorithms such as GCD computation.

ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation under Grant No. MCS 76-15035.

REFERENCES

[1] BROWN, W.S, The Subresultant PRS Algorithm, TOMS 4 (1978) 237-249

[2] MOSES, J and YUN, D.Y.Y., The EZ GCD Algorithm, Proceedings of ACM 73 (1973) 159-166

[3] BROWN, W.S., On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors, Journ A.C.M. 18 (1971) 478-504

[4] COLLINS, G.E., Subresultants and reduced polynomial remainder sequences, Journ A.C.M. 14 (1967) 128-142

[5] HEARN, A.C., An Improved Non-modular Polynomial GCD Algorithm, SIGSAM Bulletin, ACM 23, New York, (1972) 10-15

[6] HEARN, A.C., REDUCE 2 - A System and Language for Algebraic Manipulation, Proc. of Second Symposium on Symbolic and Algebraic Manipulation, International Hotel, Los Angeles (1971) 128-133

[7] HEARN, A.C., A Mode Analyzing Algebraic Manipulation Program, Proceedings of ACM 74, ACM, New York (1974) 722-724

[8] GRISS, M.L., The Definition and Use of Data-Structures in REDUCE, Proc. SYMSAC 76, ACM, New York (1976) 53-59

[9] HEARN, A.C., The Structure of Algebraic Computations, Proc. of the Fourth Colloquium on Advanced Comp. Methods in Theor. Physics, St. Maximin, France, (1977) 1-15.

APPENDIX

In this Appendix, we present some computing time comparisons (in seconds) for the three algorithms we discuss in this paper. We compare these algorithms for a set of six benchmarks with the computing times for the three seminal algorithms discussed by Moses and Yun. Although there may be other cases which differentiate the algorithms better, we feel that the current set will suffice until better benchmarks are developed.

Our algorithms have been implemented in REDUCE and run on a DECsystem 2040 in a 100K word heap. Moses and Yun's examples were run on a PDP-10 with a KA-10 processor, which is roughly half the speed of our processor. The numbers quoted below for the algorithms we present are therefore TWICE the actual times on our computer in order to be a fairer comparison with the original set of numbers. However, the growth of the sequence of numbers is surely more significant than their absolute values. The basic table and the first three entries for each example are taken directly from Ref [2], although some of the comments have been omitted.

Case 1: gcd(F,G) = 1.

$$F = \left(\sum_{i=1}^{v} x_i - 1\right)\left(\sum_{i=1}^{v} x_i + 2\right);\quad G = \left(\sum_{i=1}^{v} x_i + 1\right)\left(-3x_2x_1^2 + x_2^2 - 1\right)^2$$

	v = 2	3	4	5
EZ	0.377	0.433	0.529	0.681
Red.	0.731	4.588	27.3(1)	137.3(9)
Mod.	0.217	1.248	1.938	2.692
Prim.	1.542	0.276	0.376	0.488
Basic	1.328	0.292	0.382	0.498
Full	1.608	0.276	0.376	0.488

(Numbers in parentheses indicate the number of "garbage collections" which cost approximately 3 seconds each, made during the GCD computation. This gives a fairly good indication of the free storage space consumed by intermediate expressions).

Case 2: Sparse polynomials where degrees and the number of variables increase with v.

$$D = \sum_{i=1}^{v} x_i^v + 1;\quad F = D\cdot\left(\sum_{i=1}^{v} x_i^v - 2\right);\quad G = D\cdot\left(\sum_{i=1}^{v} x_i^v + 2\right)$$

	v=2	3	4	5	6	7
EZ	0.465	0.556	0.767	1.062	1.436	1.845
Red.	0.121	0.463	0.870	1.512	2.449	3.727
Mod.	0.629	6.291	127.1(9)	---	---	---
Prim.	0.172	0.266	0.382	0.510	0.688	0.882
Basic	0.162	0.248	0.378	0.520	0.632	0.812
Full	0.158	0.246	0.350	0.512	0.628	0.810

D and F as above; $G = D\cdot\left(\sum_{i=1}^{v} x_i^{v-1} + 2\right)\cdot$

	v=2	3	4	5	6	7
EZ	0.490	0.826	1.257	1.838	2.708	3.833
Red.	0.330	5.686	136.0(7)	---	---	---
Mod.	0.714	7.810	164.3(13)	---	---	---
Prim.	0.482	1.464	5.894	30.706(1)	136.236(4)	---
Basic	0.454	1.176	5.742	22.514	111.590(3)	---
Full	0.450	1.340	5.264	23.042	120.834(4)	---

Case 3: Quadratic GCD with quadratic factors.

$$D = x_2^2 x_1^2 + \sum_{i=3}^{v} x_i^2 + 1;$$

$$F = D\cdot(x_2 x_1 + \sum_{i=3}^{v} x_i + 2)^2; \quad G = D\cdot(x_1^2 - x_2^2 + \sum_{i=3}^{v} x_i^2 - 1)\cdot$$

	v=2	3	4	5
EZ	1.279	2.880	5.055	8.504
Red.	0.552	5.982	56.6(2)	433.3(30)
Mod.	1.433	6.668	38.6(2)	204.9(14)
Prim.	0.776	1.398	1.974	2.720
Basic	1.690	4.940	13.506	33.234
Full	0.920	1.586	2.182	3.010

Case 4:

$$F = U^p V^q \text{ and } G = U^q V^p$$

$$U = x_1 - x_2 x_3 + 1; \quad V = x_1 - x_2 + 3x_3$$

	p,q=1,2	1,3	1,4	2,4
EZ	1.693	2.856	4.201	8.324
Red.	1.265	26.98(1)	638.2(54)	68.9(3)
Mod.	4.311	8.601	18.5(1)	30.7(1)
Prim.	1.108	4.052	14.366	10.65(1)
Basic	0.840	5.536	19.076	89.460(5)
Full	0.886	3.576	12.150	10.45(1)

Case 5: Dense quadratics with linearly dense GCD.

$$D = \prod_{i=1}^{v} (x_i+1)-3; \quad F = D\cdot\prod_{i=1}^{v} (x_i-2); \quad G = D\cdot\prod_{i=1}^{v} (x_i+2)\cdot$$

	v=2	3	4	5
EZ	0.957	3.185	12.055	55.3(3)
Red.	0.457	2.865	20.912	229.2(12)
Mod.	0.658	3.019	12.600	62.2(4)
Prim.	0.766	3.320	12.588	57.34(3)
Basic	0.766	3.412	12.158	51.55(2)
Full	0.760	3.136	12.874	55.6(3)