PROBABILISTIC ALGORITHMS FOR SPARSE POLYNOMIALS

Richard Zippel*
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Mass. 02139/USA

1. Introduction.

Modular approaches [1] to algebraic algorithms, such as the GCD, have been very useful in cases where there are very few variables. These algorithms have, unfortunately, an exponential worst case behavior since they need as many as $(d+1)^v$ independent evaluations for a problem with v variables of degree d in each variable. The Hensel lemma was successfully used in many GCD and factorization problems [3, 4, 7, 9, 10] when the problems were sparse. The Hensel lemma approach exhibits exponential behavior at "bad zero" cases that correspond to a zero derivative in the Newton's method analogue. This occurs when substituting zero for one or more variables destroys too much information and reduces the corresponding Jacobian to zero. In such cases it is common to make a linear substitution, such as $y+3$ for y, in order to avoid the bad point. The substitution, however, causes a large growth in the size of the revised problem. Thus the Hensel lemma based algorithms tend to run out of space relatively early on bad zero problems.

This paper discusses a probabilitistic technique for avoiding the exponential behavior of the modular and Hensel algorithms. This technique's expected running time is a polynomial in the number of terms. Since the results for GCD and factorization can be checked by division, one is guaranteed to obtain the correct answer, if need be, by performing the calculation twice. The probability of getting incorrect results can be made so low, however, that no such backtracking has been required in any of our tests so far. As expected, the experimental results of the algorithms verify the fact that it is exponentially faster than any of the existing algorithms in their worst cases, and its performance is a polynomial function of the size of the final answer in all cases. The probabilistic algorithm presented here will be a variation of the modular GCD algorithm. In [11], we present a formulation of Hensel's lemma that is somewhat more general than the one in current use and our probabilistic analogue to it. Here, we shall only present the modular algorithm. In [11] we shall also discuss how our ideas can be used in computing determinants, resultants and solutions of both linear and non-linear equations. Except in the latter case, it is relatively difficult to check the answers, so a small probability of error is possible. But as our analysis shows, that probability can be made as low as one pleases. In [11], we also discuss the use of our main idea in solving the "intermediate expression swell" problem in those cases where the form of the final answer is known in advance.

* This work was supported, in part, by the United States Department of Energy under Contract Number E(11-1)-3070 and by the National Aeronautics and Space Administration under Grant NSG 1323.

We should also note that Paul Wang, in his Enhanced EZ GCD algorithm [8], uses ideas similar to those we use. Although the EEZ GCD algorithm will often run faster than ours, there has been no analysis of Wang's heuristics that indicates they are effective in all cases. There also seems to be empirical evidence that the EEZ GCD algorithm can still suffer from the "bad zero" problem. Furthermore, it is not clear how to extend his approach to problems other than GCD or factorization.

The basic idea of our probabilistic approach is as follows: We substitute randomly chosen, large integers for all but one variable in the problem. The solution is built up by interpolating for one variable at a time. Our main probabilistic assumption is that when a coefficient has been determined to be zero somewhere in the interpolation process it is assumed to be zero everywhere. Thus, one need never compute more terms then there will actually be in the answer. The algorithm resorts to solving t linear equations at each level, where t is the number of terms at that level. Thus its cost is asymptotically cubic in the number of terms.

2. Sparse Modular Algorithm.

All modular algorithms have basically the same form—a polynomial is interpolated from its value at a number points. We will call this polynomial the *goal polynomial* of the algorithm. The goal polynomial is assumed to involve v variables. Each variable appears to no higher degree than d in the goal polynomial. The goal polynomial will be denoted by $P(X_1, \ldots, X_v)$.

There are $(d+1)^v$ independent coefficients in P. An algorithm that has no probabilistic aspects needs at least $(d+1)^v$ "points" worth of information to determine these coefficients. Just looking at these points requires time exponential in the number of variables. Throughout this section P is assumed to be sparse, and has t terms ($t \ll (d+1)^v$).

2.1. Overview of Sparse Modular Algorithm.

The sparse modular algorithm begins by choosing a starting point for the interpolation, $(x_{10}, \ldots, x_{v0})$. It then produces the sequence of polynomials,

$$\begin{aligned} P_1 &= P(X_1, x_{20}, \ldots, x_{v0}), \\ P_2 &= P(X_1, X_2, x_{30}, \ldots, x_{v0}), \\ &\vdots \qquad\qquad \vdots \\ P_v &= P(X_1, X_2, \ldots, X_v). \end{aligned}$$

Note that P_1 is a univariate polynomial in X_1. The coefficient of X_1^k in P is a polynomial $f_k(X_2, \ldots, X_v)$. If P is sufficiently sparse there will be certain powers of X_1 that do not appear in P_1. Assume that the X_1^k term is one of those terms that is not present. There are two possible explanations why X_1^k did not appear in P_1. Either f_k is identically zero or $f_k(x_{10}, \ldots, x_{v0})$ is equal to zero. If the starting point $(x_{10}, \ldots, x_{v0})$ is chosen at random then the probability that $f_k(x_{10}, \ldots, x_{v0})$ is zero is extremely small. Thus the probability that f_k

is identically zero is quite large. The key idea in this algorithm is to assume that X_1^k does not appear in P; i.e., f_k is identically zero. Thus it is assumed that the coefficient of every monomial involving X_1^k is known, and that it is zero.

This information is used to construct P_2. The same reasoning can be applied to each monomial in X_1 and X_2 that does not appear in P_2. Since there are at most t terms in any of the P_i, almost all of the terms will be zero the number of coefficients that we don't know is small.

We will demonstrate this algorithm when P is a polynomial in 3 variables, $P(X, Y, Z)$. As usual, we assume that P is a sparse polynomial with t terms ($t \ll (d+1)^v$, $v = 3$). Whenever we say "pick x_i" we will mean pick an integer x_i randomly from a set $\mathcal{S}$ that has at least B distinct elements.* Pick y_0 and z_0 randomly. We now pick $x_0, \ldots, x_d$ and examine the values of P at the points (x_i, y_0, z_0). These may be interpolated to give a univariate polynomial in X, namely $P(X, y_0, z_0)$. So far nothing probabilistic has entered the algorithm.

We now assume that if some power of X had a zero coefficient in $P(X, y_0, z_0)$ it will have a zero coefficient in $P(X, Y, Z)$. Pick a y_1. From $P(X, y_0, z_0)$ we know that a number of the coefficients of $P(X, y_1, z_0)$ are zero. The only coefficients that need to be determined are the non-zero ones. There can be no more than t of these unknown coefficients. They can be determined by solving a system of linear equations. Only the values of $P(x_0, y_1, z_0), \ldots, P(x_t, y_1, z_0)$ will be needed to set up this system of equations.

This procedure may be repeated until we have determined the sequence of polynomials $P(X, y_0, z_0), \ldots, P(X, y_d, z_0)$. Pick a monomial in X which appears in each of these polynomials. For simplicity we will assume that it is the linear term. The linear term (in X) of $P(X, Y, z_0)$ is a polynomial in Y of degree at most d. Call this polynomial $f(Y)$. From the $d+1$ polynomials we have computed we can determine the values of $f(Y)$ at $y_0, \ldots, y_d$. Again using the usual interpolation methods we can determine $f(Y)$ from this information. Doing this with all the coefficients of the $P(X, y_i, z_0)$ we can determine $P(X, Y, z_0)$.

Now that we have $P(X, Y, z_0)$, it is only natural to try to compute $P(X, Y, z_1)$ for a new z_1. This can be done in a manner almost identical with that used earlier. We know that the monomials which appear in $P(X, Y, z_1)$ will have non-zero coefficients in $P(X, Y, Z)$. We assume that none of the X^iY^j monomials in P are missing. There are at most t of these monomials, and thus at most t unknown coefficients to be determined. Picking t pairs of values $(x_1, y_1), \ldots, (x_t, y_t)$ and computing $P(x_i, y_i, z_1)$ we can set up a system of linear equations in the unknown coefficients. Solving this system we have $P(X, Y, z_1)$. Repeating this procedure we will finally determine $P(X, Y, z_d)$. By repeating the standard interpolation scheme we will finally arrive at $P(X, Y, Z)$.

There are two essentially different types of interpolation going on in this algorithm. The first time we try to generate a polynomial in X, it is not known what its structure is and thus the interpolation is preformed as if the polynomial were dense. This we call a *dense* interpolation. (Actually the polynomial in X can be read off from its values using the Lagrange interpolation formula, but this gives only a slight increase in efficiency.) Now a

*The set $\mathcal{S}$ is usually chosen to be the interval $[0, B-1]$. Thoughout this section lower case symbols will denote integers chosen at random while uppercase symbols will be reserved for variables.

number of *sparse* interpolations are done for different values of Y to get more polynomials in X. The coefficients are then combined via a dense interpolation to give polynomials in Y. The algorithm proceeds in this manner. The first polynomial produced involving a particular variable is done via a dense interpolation. The structure determined by the dense interpolation is then used to produce a skeleton for the polynomial. This skeleton is used as the basis for a series of sparse interpolations which are done to set up the points for a new variable.

2.2. General Formalism of Sparse Modular Algorithm.

In this section we will present a precise form of the sparse modular algorithm that will also aid in the analysis of the algorithm. Algorithm D makes no assumptions about the sparsity of the goal polynomial. It uses the Chinese remainder algorithm to produce a univariate polynomial over a field. This is the *dense* lifting stage mentioned in the previous section.

Algorithm D. Given two sets of rational integers $\{p_1, \ldots, p_k\}$ and $\{m_1, \ldots, m_k\}$, it returns a polynomial $f(x)$ such that $f(p_i) = m_i$ for $1 \leq i \leq k$.

D1. [Initialize] Set $f(x) \leftarrow m_1$, $q(x) \leftarrow (x - p_1)$.

D2. [Loop] For $i = 2, \ldots, k$ do step D3.

D3. [Determine new f] Set $f(x) \leftarrow f(x) + q(p_i)^{-1} q(x)(m_i - f(p_i))$ and $q(x) \leftarrow (x - p_i)q(x)$.

D4. [End] Return $f(x)$.

It is important to note that even if the goal polynomial for algorithm D is very sparse the intermediate results can be completely dense. The full sparse modular algorithm alternates between stages of dense interpolations using algorithm D above, and stages of sparse interpolation in algorithm S below.

The sparse interpolation algorithm needs a data structure to indicate which terms are known to be zero. Since there are fewer terms which are likely to have nonzero coefficients than terms with zero coefficients, we will keep track of the nonzero terms. A monomial of the form $X_1^{e_1} \cdots X_v^{e_v}$ will be represented by the v-tuple $(e_1, \ldots, e_v)$. A *skeletal polynomial* S, is understood to be a set v-tuples where each element of S represents a nonzero term in the goal polynomial.

After a skeletal polynomial is produced we will want to determine what its coefficients are. This will be done by solving a system of linear equations. To simplify the notation a bit we will adopt the following convention. Assume a skeletal polynomial S contains t terms. We will assume that each skeletal polynomial has associated with it t symbols which will represent the coefficients of the monomials given by S. Denote these symbols by $s_1, \ldots, s_t$ where the subscript, i, is associated with the exponent vector $(e_{i1}, \ldots, e_{iv})$. Then we define

$$S(a_1, \ldots, a_v) = s_1 a_1^{e_{11}} \cdots a_v^{e_{1v}} + s_2 a_1^{e_{21}} \cdots a_v^{e_{2v}} + \cdots + s_t a_1^{e_{t1}} \cdots a_v^{e_{tv}}.$$

The sparse modular algorithm can be specified as follows.

Algorithm S takes a set of variables $\{X_1, \ldots, X_v\}$, a degree bound d, a function $F(X_1, \ldots, X_v)$ and a starting point $(a_1, \ldots, a_v)$ as arguments. It is assumed that the values F returns are the values of some polynomial of at most v variables and of degree at most d in each variable. The starting point is assumed to be a good stating point. The algorithm returns a polynomial $P(X_1, \ldots, X_v)$, where each variable occurs to degree no more than d and $P(b_1, \ldots, b_v) = F(b_1, \ldots, b_v)$ for all integers b_i.

S1. [Initialize] Set $S \leftarrow \{(0)\}$ and $p_0 \leftarrow a_0$.

S2. [Loop over variables] For $i = 1$ thru v do S3 thru S8.

S3. [Iterate d times] For $j = 1$ thru d do S4 thru S7.

S4. [Initial linear equations] Pick r_j, set L to the empty list, set t to the length of S.

S5. [Iterate t times] For $k = 1$ thru t do S6.

S6. [Set up liner equations] Pick a random $(i-1)$-tuple Λ_k, and add the the linear equation $S(\Lambda_k) = F(\Lambda_k, r_j, a_{j+1}, \ldots, a_v)$.

S7. [Solve] Solve the system of equations L and merge the solution with S to produce a polynomial $p_j(X_1, \ldots, X_{i-1})$.

S8. [Introduce X_i] For each monomial in S pass the corresponding coefficients from $p_0, \ldots, p_d$ and $a_i, r_1, \ldots, r_j$ to algorithm D. This will produce t polynomials which can be merged with S. Set p_0 to this new polynomial and S to its skeletal polynomial.

S9. [Done] Return p_0.

There is one point at which caution should be exercised in implementating this procedure. The first time through the i loop the linear equations which are set up will be trivial since there is only one unknown. There is a chance that the linear equations that are developed will not be independent. If this happens then it is necessary to run step S6 until sufficiently many independent equations are produced.

3. Analysis and Timings.

Probabilistic algorithms are rather new in algebraic manipulation. Other probabilistic algorithms are discussed in [5,6]. In this section we first define what is meant by a "good starting point." The probability that a random point is good is then determined. This probability is very small and can easily be made even smaller. Then the running time of the algorithms of section 2 are analyzed. Finally a number of sample problems are presented to compare the analysis, the actual running time and the running time of several competing algorithm including the EZGCD algorithm.

3.1. Good and Bad Points.

Assume the goal polynomial is $P(X_1, \ldots, X_v)$ and the starting point is $\vec{a} = (a_1, \ldots, a_v)$. The polynomials which are produced by the sequence of dense iterations is

$$P(X_1, a_2, \ldots, a_v), P(X_1, X_2, a_3, \ldots, a_v), \ldots, P(X_1, X_2, \ldots, X_v).$$

The entire algorithm depends upon the accuracy of the skeletal polynomials. The skeletal polynomials are extracted from the structure of the polynomials in this sequence. Thus it is important to know if $P(X_1, a_2, \ldots, a_v)$ has too few terms. This will happen if the coefficient of some X_1^k in $P(X_1, \ldots, X_v)$ is zero at $\vec{a}$. Let F_1 be the product of the nonzero coefficients of X_1^k in P for $k = 1$ thru d. If $\vec{a}$ is not a zero of F_1 then the second skeletal polynomial will be computed correctly.

Similarly if the coefficient of some monomial in X_1 and X_2 is zero at $\vec{a}$ the second skeletal polynomial will be erroneous. Define F_2 to be the product of the coefficients of nonzero monomials in X_1 and X_2 and define $F_3, \ldots, F_{v-1}$ similarly. The *auxiliary polynomial* for P is defined to be $F = F_1 F_2 \cdots F_{v-1}$. F is a polynomial in $X_2, \ldots, X_v$. The key assumption throughout this section is that our initial evaluation point is not a zero of this polynomial. A point at which F is non-zero is called a *good point*. F is the auxiliary polynomial which was mentioned earlier. Since all bad points satisfy $F = 0$ they form a variety of codimension 1. Thus almost all points in affine $v-1$ space are good.

Each of the F_j is the product of no more than t polynomials. Thus the degree of X_i in F_j is bounded by dt and in F by dvt. The following theorem gives the probability that a point chosen from a set of a given size will be bad for a polynomial of degree D in v variables.

Theorem 1. *Let $f \in \mathbb{Z}[X_1, \ldots, X_v]$ and the degree of f in X_i be bounded by D. Let $N_v(B)$ be the number of zeroes of f, $(x_1, \ldots, x_v)$ such that $x_i \in \mathfrak{I}$ (a set with B elements, $B \gg D$). Then $N_v(B) \leq B^v - (B-D)^v$.*

Proof: There are at most D values of x_v which zero f identically. So for any of the D values of x_v and any value for the other x_i, f is zero. This comes to DB^{v-1}. For all other $B-D$ values of x_v we have a polynomial in $v-1$ variables. The polynomial can have no more than $N_{v-1}(B)$ zeroes. Therefore,

$$N_v(B) \leq DB^{v-1} + (B-D)N_{v-1}(B).$$

Let $N_v = (B-D)^{v-1} f_v$. The resulting equation is easily solved and the theorem follows directly.

This bound is actually attained by the polynomial

$$f(x_1, \ldots, x_v) = \prod_{i=1}^{D}(x_1 - i) \cdots \prod_{i=1}^{D}(x_v - i).$$

This polynomial is dense in all of its variables. One would expect a much tighter bound to hold for sparse polynomials.

Each of the F_i is the product of, at most, t terms and each term is of degree, at most, d. There are $v-1$ of these polynomials, so the maximum possible degree of F is $(v-1)td$. There are only $v-1$ variables in F. There are B^v points in the set $\mathfrak{I} \times \cdots \times \mathfrak{I}$. Applying

the theorem to F the probability that a point chosen at random will be a zero of F is

$$\begin{aligned}\frac{N_{v-1}(B)}{B^{v-1}} &= \frac{B^{v-1}-(B-D)^{v-1}}{B^{v-1}} \\ &= 1-\left(1-\frac{D}{B}\right)^{v-1} \\ &\leq \frac{v(v-1)td}{B} \leq \frac{v^2td}{B}.\end{aligned}$$

At worst the number of terms in the goal polynomial will be $(d+1)^v$. So a worst case bound for the probability that a point will be a zero of F, and thus a bad point is

$$\frac{v^2d(d+1)^v}{B}.$$

If we wanted to make the probability of choosing a bad point be at most 10^{-30} we would have

$$\begin{aligned}B &\geq 10^{30}v^2(d+1)^{v+1} \\ \log B &\geq 69+2\log v+(v+1)\log(d+1).\end{aligned}$$

Notice that the size of the numbers which are used is about $v\log d$. Thus each arithmetic operation will take polynomial time. Since there are only a polynomial number of arithmetic operations the algorithm's expected running time is polynomial.

3.2. Analysis.

Throughout this section we will assume that all arithmetic can be done in unit time, the goal polynomial involves v variables and no variable appears to degree more than d in the goal polynomial.

We will make a number of crude assumptions in analyzing Algorithm S. We assume that cost of evaluating F is constant and requires C arithmetic operations. We will also assume that the number of terms of $P(X_1, a_2, \ldots, a_v)$ is t_1, of $P(X_1, X_2, a_3, \ldots, a_v)$ is t_2 and so on; t_v is equal to t.

Each monomial contained in S is a product of $i-1$ terms, and each term is exponentiated to degree, at most, d. Evaluation of a monomial will thus cost $(i-1)\log d$ operations. There are no more than t_{i-1} terms in S, so step S6 will take about $C+(i-1)t_{i-1}\log d$ operations. Step S6 will be iterated t_{i-1} times to produce the each set of linear equations. Thus it will cost $Ct_{i-1}+(i-1)t_{i-1}^2\log d$ operations to produce the system of linear equations.

There will be t_{i-1} independent linear equations to be solved. Using straight forward algorithms this will take about $c_1t_{i-1}^3$ operations. Steps S5 thru S7 will be executed d times for each variable so it will cost

$$Cdt_{i-1}+(i-1)dt_{i-1}^2\log d+c_1dt_{i-1}^3$$

operations to produce the polynomials, $p_1, \ldots, p_{t_{i-1}}$.

There will be t_{i-1} terms in each of these polynomials, so algorithm D will be run t_{i-1} times. Each invocation of algorithm D will require about $c_2 d^2$ operations. Adding this mess up and summing from $i = 1$ thru v we get

$$\sum_{i=1}^{v} \left((c_2 d^2 + Cd) t_{i-1} + (i-1) d t_{i-1}^2 \log d + c_1 d t_{i-1}^3 \right)$$

We need to make some assumptions about the structure of t_i to get anything meaningful out of this. We will assume that the ratio of terms t_i / t_{i-1} is a constant, k. Doing this we get

$$c_1 d \frac{(k^{3v}-1) t_0^3}{(k^3-1)} + \frac{d \log d t_0^2}{(k^2-1)^2} \left[k^{2v} \left(k^2 (v-1) - v \right) + k^2 \right] + c_2 d^2 \frac{(k^v - 1) t_0}{(k-1)}$$

Despite appearances to the contrary this expression is not exponential in v. Remember that $k^v t_0 = t$. There are two special cases of this formula that are of interest. If k is large when compared with 1, we can ignore the small terms involving k and get

$$c_1 d \frac{t^3}{k^3} + d \log d v \frac{t^2}{k^2} + c_2 \frac{t}{k}.$$

If k is very close to 1, then $t_0 = t$ and we get

$$c_1 d v t^3 + d v^2 t^2 \log d + c_2 d^2 v t.$$

In both of these cases the dominant behavior is $O(t^3)$, assuming $t \gg d$ or v. This is clearly not exponential in the number of variables (unless t is) which is unlike any other modular algorithm.

3.3. Timings.

Here we present a few sample timings and compare them with our estimates from the previous section. A more detailed analysis and further examples are contained in [11].

The first example was chosen to show the sparse modular GCD algorithm at its best. Nine monomials were chosen at random and combined to produce three polynomials with 3 terms each. One was multiplied by the other two to give two polynomial of 9 terms. These two polynomials were used as the input to the GCD routines. The number of variables ranged up to 10 and the degree of each variable was less than 3. The following table gives the computation times for the EZGCD algorithm, the Modular algorithm, the Reduced algorithm, Wang's new EEZGCD algorithm and finally the Sparse Modular algorithm. These timings were done on a DEC KL-10 using MACSYMA [2]. The polynomials used are

contained in the appendix. The asterisks indicate that the machine ran out of space.

v	EZ	Modular	Reduced	EEZ	Sparse Mod
1	.036	.047	.047	.036	.040
2	.277	.275	.216	.377	.160
3	.431	.920	.478	.522	.381
4	1.288	7.595	2.027	.742	.842
5	3.128	65.280	*	1.607	1.825
6	*	483.700	*	1.897	3.364
7	*	2409.327	*	1.715	4.190
8	*	*	*	*	4.534
9	*	*	*	*	4.006
10	*	*	*	*	8.202

As expected the modular algorithm ran in exponential time. Both the EZ and the Reduced algorithms ran out of storage. This example was carefully designed so that all the GCD's were bad zero problems for the EZ algorithm. When these problems were run a LISP machine with 30 million words of address space the exponential behavior of the EZ algorithm was evident.

4. Conclusions.

In this paper we have tried to demonstrate how sparse techniques can be used to increase the effectiveness of the modular algorithms of Brown and Collins. These techniques can be used for an extremely wide class of problems and can applied to a number of different algorithms including Hensel's lemma. We believe this work has finally laid to rest the bad zero problem.

Much of the work here is the direct result of discussion with Barry Trager and Joel Moses whose help we wish to acknowledge.

5. Appendix.

This appendix lists the polynomials that were used to test the various GCD algorithms in section 3. The d_i polynomials are the GCDs which are computed, the f_i and g_i the cofactors. The polynomials that were fed to the various GCD routines were $d_i f_i$ and $d_i g_i$.

$$d_1 = x_1^2 + x_1 + 3$$
$$f_1 = 2x_1^2 + 2x_1 + 1$$
$$g_1 = x_1^2 + 2x_1 + 2$$

$$d_2 = 2x_1^2x_2^2 + x_1x_2 + 2x_1$$
$$f_2 = x_2^2 + 2x_1^2x_2 + x_1^2 + 1$$
$$g_2 = x_1^2x_2^2 + x_1^2x_2 + x_1x_2 + x_1^2 + x_1$$

$$
\begin{aligned}
d_3 &= x_2^2x_3^2 + x_2^2x_3 + 2x_1^2x_2x_3 + x_1x_3\\
f_3 &= x_3^2 + x_2^2x_3 + x_1^2x_2x_3 + x_1x_3 + x_1^2x_2^2\\
g_3 &= x_2x_3 + 2x_1x_3 + x_3 + x_1
\end{aligned}
$$

$$
\begin{aligned}
d_4 &= x_1^2x_4^2 + x_2^2x_3x_4 + x_1^2x_2x_4 + x_2x_4 + x_1^2x_2x_3\\
f_4 &= x_1x_2x_3^2x_4^2 + x_1x_3^2x_4^2 + x_1x_4^2 + x_4^2 + x_1x_3x_4\\
g_4 &= x_1x_3^2x_4^2 + x_3^2x_4^2 + x_4^2 + x_1x_2^2x_3x_4 + x_1x_2^2
\end{aligned}
$$

$$
\begin{aligned}
d_5 &= x_1^3x_2^2x_3^2x_4x_5^2 + x_1x_2^2x_5^2 + x_1^3x_3x_4^2x_5 + x_1^3x_2x_3^2x_4x_5 + x_1^2x_2x_3^2x_4^2\\
f_5 &= x_1x_2^2x_5^2 + x_1x_2x_3^2x_4x_5 + x_1x_2x_3^2x_4^2 + x_1x_2^2x_4^2 + 1\\
g_5 &= x_1x_3^2x_4x_5^2 + x_2x_5^2 + x_1x_2x_4x_5 + x_2x_5 + x_1x_2x_3x_4^2
\end{aligned}
$$

$$
\begin{aligned}
d_6 &= x_1x_2x_4^2x_5^2x_6^2 + x_1x_2^2x_3^2x_4x_5^2x_6^2 + x_1^2x_3x_6^2 + x_1^2x_2x_3^2x_4x_5^2x_6 + x_1^2x_3x_5x_6\\
f_6 &= x_1^2x_2x_4x_5^2x_6^2 + x_1x_3x_5^2x_6^2 + x_1x_2^2x_6^2 + x_1^2x_2^2x_3^2x_5x_6 + x_1x_3^2x_4x_5\\
g_6 &= x_2^2x_3^2x_4x_5^2x_6 + x_1x_4^2x_5x_6 + x_2^2x_3^2x_4x_5x_6 + x_1x_2^2x_3x_4^2x_6 + x_1^2x_3x_5^2
\end{aligned}
$$

$$
\begin{aligned}
d_7 &= x_1x_2^2x_4^2x_6^2x_7^2 + x_1^2x_3x_4x_6^2x_7^2 + x_3^2x_4^2x_7^2 + x_1^2x_2x_4^2x_6 + x_3x_4x_5^2\\
f_7 &= x_1^2x_2x_4^2x_5x_6^2x_7^2 + x_1x_2x_3x_6x_7 + x_3x_4^2x_5^2x_7 + x_1x_4^2x_5^2x_7 + x_1^2x_2x_3x_4^2x_5x_6\\
g_7 &= x_1x_3x_5x_6^2x_7^2 + x_2^2x_3^2x_4^2x_5x_6x_7^2 + x_4x_6x_7^2 + x_1^2x_2x_3x_5x_6x_7 + x_1^2x_3^2x_4x_5^2
\end{aligned}
$$

$$
\begin{aligned}
d_8 &= x_2^2x_4x_5x_6x_7x_8^2 + x_1^2x_2x_3^2x_4^2x_6^2x_7^2x_8 + x_1^2x_3x_4^2x_6^2x_7^2 + x_1^2x_2^2x_3^2x_4x_5^2x_6x_7^2 + x_2^2x_4x_6\\
f_8 &= x_1^2x_2^2x_3x_4^2x_5x_6^2x_8^2 + x_2x_5x_6^2x_8^2 + x_1^2x_2^2x_3^2x_4^2x_6^2x_7^2x_8 + x_1^2x_3^2x_4x_5^2x_7^2x_8 + x_1x_2^2x_3^2x_5^2x_7\\
g_8 &= x_1x_4^2x_5x_6x_7x_8^2 + x_1x_2^2x_4^2x_5^2x_6^2x_8 + x_1^2x_2x_3x_4^2x_6^2x_8 + x_1^2x_2^2x_3^2x_4x_5^2x_8 + x_1x_2x_4^2x_5^2
\end{aligned}
$$

$$
\begin{aligned}
d_9 &= x_1^2x_3^3x_4x_6x_8x_9^2 + x_1x_2x_3x_4^2x_5^2x_8x_9 + x_2x_3x_4x_5^2x_8x_9 + x_1x_3^3x_4^2x_5^2x_6^2x_7x_8^2 + x_2x_3x_4x_5^2x_6x_7x_8^2\\
f_9 &= x_1^2x_2^2x_3x_7^2x_8x_9 + x_2^2x_9 + x_1^2x_3x_4^2x_5^2x_6x_7^2 + x_4^2x_5^2x_7^2 + x_3x_4^2x_6x_7\\
g_9 &= x_1^2x_2x_4x_5x_6x_7^2x_8^2x_9^2 + x_1^2x_2x_3x_5x_6^2x_7^2x_8x_9^2 + x_1^2x_3x_4x_6x_7^2x_8x_9 + x_1^2x_2^2x_6x_8^2 + x_2^2x_4x_5x_6^2x_7
\end{aligned}
$$

$$
\begin{aligned}
d_{10} = {} & x_1x_2^2x_4^2x_8x_9^2x_{10}^2 + x_2^2x_4x_5^2x_6x_7x_9x_{10}^2 + x_1^2x_2x_3x_5^2x_7^2x_9^2 + x_1x_3^2x_4^2x_7^2x_9^2 + x_1^2x_3x_4x_7^2x_8^2\\
f_{10} = {} & x_1x_2x_3^2x_4x_6x_7x_8x_9^2x_{10}^2 + x_2^2x_3^2x_4^2x_6^2x_9x_{10}^2 + x_1x_2^2x_3^2x_4x_5x_6x_7x_8^2x_9^2x_{10}\\
& + x_1^2x_2x_4^2x_5^2x_8^2x_9^2x_{10} + x_3x_4^2x_5x_6x_7^2x_9x_{10}\\
g_{10} = {} & x_1x_2^2x_3^2x_5^2x_6^2x_7x_8x_9^2x_{10}^2 + x_3x_8x_9^2x_{10}^2 + x_1x_2^2x_3x_4x_5^2x_6^2x_8^2x_9x_{10}\\
& + x_1x_3x_6x_7x_8x_{10} + x_1^2x_5^2x_6^2x_7x_9^2
\end{aligned}
$$

References

1. S. Brown, "On Euclid's Algorithm and the Computation of Polynomial Greatest Divisors," *J. ACM* 18, 4 (1971), 478–504.
2. MATHLAB Group, *MACSYMA Reference Manual—version 9*, Laboratory for Computer Science, Massachusetts Institute of Technology, (1978).
3. J. Moses and D. Y. Y. Yun, "The EZGCD algorithm," *Proceedings of ACM Nat. Conf.* (1973), 159–166.
4. D. R. Musser, "Multivariate Polynomial Factoring," *J. ACM* 22, 2 (1975), 291–308.
5. M. O. Rabin, "Probabilistic Algorithms," *Algorithms and Complexity—New Directions and Recent Results (J. F. Traub Ed.)*, Acad. Press, New York, (1976), 21–39.
6. R. Solovay and V. Strassen, "A Fast Monte-Carlo Test for Primality," *SIAM J. of Comp.* 6, 1 (1977).
7. P. S.-H. Wang and L. P. Rothschild, "Factoring Multivariate Polynomials over the Integers," *Math. Comp.* 29, (1975), 935–950.
8. P. S.-H. Wang, "An Improved Multivariate Polynomial Factoring Algorithm," *Math. Comp.* 32, (1978), 1215–1231.
9. D. Y. Y. Yun, *The Hensel Lemma in Algebraic Manipulation*, Ph. D. thesis, Massachusetts Institute of Technology, (1974).
10. H. Zassenhaus, "On Hensel Factorization I," *J. Number Theory* 1, (1969), 291–311.
11. R. E. Zippel, *Probabilistic Algorithms for Sparse Polynomials*, Ph. D. thesis, Massachusetts Institute of Technology, (1979).