On Square-Free Decomposition Algorithms

by

David Y. Y. Yun
Mathematical Sciences Department
IBM Thomas J. Watson Research Center
Yorktown Heights, N. Y.

1. Introduction

Given a polynomial P in R[x] where R is a unique factorization domain, P is said to be square-free if P has no divisor (or factor) of multiplicity ≥ 2. The problem of finding the square-free (abbreviated as SQFR) decomposition of P amounts to finding polynomials $P_1, P_2, \ldots, P_k$ such that $P = P_1 P_2^2 \ldots P_k^k$, $P_k \neq 1$, each P_i is square-free, and gcd $(P_i, P_j)=1$, $i \neq j \leq k$.

Square-free decompositions of polynomials have many uses. Among them, the most important ones are in polynomial factorization and partial fraction decomposition of rational functions. These are, in turn, indispensable tools of rational function integrations [MOS67], [MOS71], [HOR71], [RIS69]. In addition, the SQFR decomposition provides a useful and inexpensive alternative toward the computation of multiplicative basis in the radical simplification problem [FAT72], [C&F76]. Horowitz [HOR71] made a quite complete survey and analysis on computing square-free decompositions, partial fraction decompositions and rational function integrations for the univariate case. Many existing symbolic manipulation systems such as MACSYMA [MAC73], SAC-1 [HOR69], and SCRATCHPAD [GJY75] also include built-in functions for computing square-free decompositions of multivariate polynomials. Musser [MUS71] presented an improved version of the SQFR decomposition algorithm by Horowitz. In this paper, we will give three different (including revamped versions of Horowitz's and Musser's) algorithms for computing the SQFR decomposition of polynomials in R[x]. Some algorithm analysis will be carried out to show the (asymptotic) superiority of the algorithm we propose (last of the three). The basic idea for this algorithm is, however, surprisingly simple and can be easily realized to be computationally more efficient. Finally, two versions of multivariate SQFR decomposition algorithms using the principles of modular arithmetic and the Hensel Construction will be presented. One of them stands out as it takes the full advantage of all the techniques. Both versions do establish that the cost of computing the SQFR decomposition is not much more than that of computing the greatest common divisor (GCD) of the polynomial and its derivative.

2. Three Direct Algorithms

First we state and prove a fundamental fact:

Theorem 1: If $P = P_1 P_2^2 \ldots P_k^k$ is primitive in x then $\gcd(P, dP/dx) = P_2 P_3^2 \ldots P_k^{k-1}$.

Proof: Let $D = \gcd(P, dP/dx)$. For an arbitrary P_i write $P = P_i^i Q$, then $dP/dx = i\, P_i^{i-1} (dP_i/dx)\, Q + P_i^i (dQ/dx)$, i.e., P_i^{i-1} divides dP/dx. Thus $\prod_{i=2}^{k} P_i^{i-1}$

divides D since the P_i's are pairwise relatively prime. On the other hand, $dP/dx = \left(\prod_{i=2}^{k} P_i^{i-1}\right)\left(\sum_{i=1}^{k} i(dP_i dx) \prod_{j\neq i} P_j\right)$

Let S be a divisor of some P_j, then S does not divide any other P_i for $i\neq j$ since $\gcd(P_i,P_j)= 1$ for all $i\neq j$. Because P_j is square-free and primitive w.r.t. x, $\gcd(P_j,dP_j/dx) = 1$, so that S does not divide dP_j/dx. That means S divides every term in the summation except one-- $j(dP_j/dx)$, hence S does not divide the sum.

We now present three square-free decomposition algorithms where given a primitive polynomial P(x) in R[x], the algorithms will compute square-free and pairwise relatively prime polynomials $P_i(x)$, i = 1,...,k, (some of them may equal 1) such that $P = P_1P_2^2 \ldots P_k^k$, for some $k \leq 1$ and $P_k\neq 1$.

(a) (Tobey and Horowitz):

$C_1 \leftarrow \gcd(P, dP/dx);$ $\quad C_1 = P_2P_3^2 \ldots P_k^{k-1}$

$D_1 \leftarrow P/C_1;$ $\quad D_1 = P_1P_2 \ldots P_k$

For i = 1, step 1, until $C_i = 1$, do: $\quad C_i = P_{i+1} P_{i+2}^2 \ldots P_k^{k-1}$

$\quad D_i = P_iP_{i+1} \ldots P_k$

$\quad C_{i+1} \leftarrow \gcd(C_i, dC_i/dx);$ $\quad C_{i+1} = P_{i+2} P_{i+3}^2 \ldots P_k^{k-i-1}$

$\quad D_{i+1} \leftarrow C_i/C_{i+1};$ $\quad D_{i+1} = P_{i+1} \ldots P_k$

$\quad P_i \leftarrow D_i/D_{i+1}.$

(b) (Musser):

$C_1 \leftarrow \gcd(P, dP/dx);$ $\quad C_1 = P_2P_3^2 \ldots P_k^{k-1}$

$D_1 \leftarrow P/C_1;$ $\quad D_1 = P_1 \ldots P_k$

For i = 1, step 1, until $D_i = 1$, do: $\quad C_i = P_{i+1} P_{i+2}^2 \ldots P_k^{k-1};$

$\quad D_i = P_i \ldots P_k$

$\quad D_{i+1} \leftarrow \gcd(C_i,D_i);$ $\quad D_{i+1} = P_{i+1} \ldots P_k$

$\quad C_{i+1} \leftarrow C_i/D_{i+1};$ $\quad C_{i+1} = P_{i+2} P_{i+3}^2 \ldots P_k^{k-i-1}$

$\quad P_i \leftarrow D_i/D_{i+1};$

(c) $G \leftarrow \gcd(P, dP/dx)$ $\quad G = P_2P_3^2 \ldots P_k^{k-1}$

$C_1 \leftarrow P/G$ $\quad C_1 = P_1 \ldots P_k$

$D_1 \leftarrow (dP/dx)/G - DC_1/dx$ $\quad D_1 = P_1 \frac{dP_2}{dx} \ldots P_k + 2P_1P_2 \frac{dP_3}{dx} \ldots P_k$

$\quad + \ldots +(k-1) P_1P_2 \ldots \frac{dP_k}{dx}$

For i = 1, step 1, until $C_i = 1$, do: $\quad C_i = P_iP_{i+1} \ldots P_k$

$\quad D_i = P_i(\frac{dP_{i+1}}{dx} \ldots P_k + 2P_{i+1} \frac{dP_{i+2}}{dx} \ldots P_k$

$\quad P_i = \gcd(C_i,D_i);$ $\quad + \ldots +(k-i) P_{i+1} \ldots \frac{dp_k}{dx})$

$\quad C_{i+1} = C_i/P_i;$

$\quad D_{i+1} = D_i/P_i - dC_{i+1}/dx.$

To show the validity of Algorithm (c), essentially by induction, we need to establish two facts, both are corollaries of Theorem 1.

Corollary 1.1: Let P'=dP/dx, G=gcd(P,P'), then

$$P'/G-(P/G)' = P_1(P_2'P_3 \ldots P_k + 2P_2P_3' \ldots P_k + \ldots + (k-1)\, P_2P_3 \ldots P_{k-1}P_k').$$

Proof. By the chain rule:

$$P' = P_1'P_2^2 \ldots P_k^k + P_1 2P_2'P_2\, P_3^3 \ldots P_k^k + \ldots + P_1P_2^2 \ldots P_{k-1}^{k-1}\, k\, P_k'\, P_k^{k-1}.$$

$$P'/G = P_1'P_2 \ldots P_k + 2P_1P_2'P_3 \ldots P_k + \ldots + k\, P_1P_2 \ldots P_{k-1}\, P_k'.$$

$P/G = P_1P_2 \ldots P_k$, so

$$(P/G)' = P_1'P_2 \ldots P_k + P_1\, P_2'\, P_3 \ldots P_k + \ldots + P_1 \ldots P_{k-1}\, P_k'.$$

Therefore (P'/G) - (P/G)' is as claimed. //

Corollary 1.2: $\gcd(P/G,\ (P'/G) - (P/G)') = P_1$

Proof. $P/G = P_1 \ldots P_k$ and $(P'/G)-(P/G)' = P_1Q$ where

$$Q = P_2'P_3 \ldots P_k + 2P_2\, P_3'\, P_4 \ldots P_k + \ldots + (k-1)\, P_2 \ldots P_{k-1}\, P_k'.$$

Since P_1 divides both, P_1 divides, or is a divisor of, the GCD. To see that P_1 is the GCD, it suffices to show that $\gcd(P_2 \ldots P_k, Q) = 1$. Let $P^* = P_2P_3^2 \ldots P_k^{k-1}$, then Theorem 1 provides that $G^* = \gcd(P^*, dP^*/dx) = P_3P_4^2 \ldots P_k^{k-2}$ where the corresponding cofactors are relatively prime, i.e., $\gcd(P^*/G^*,\ (dP^*/dx)/G^*) = 1$. But $P^*/G^* = P_2 \ldots P_k$ and $(dP^*/dx)/G^* = Q$. //

3. Comparison of Algorithms

For Algorithms (a) and (b), $D_i = P_iP_{i+1} \ldots P_k$ and $C_i = P_{i+1}P_{i+2}^2 \ldots P_k^{k-1}$ are computed in different ways. Algorithm (b) appears to be more efficient for the following reasons: (1) the GCD computations involve smaller inputs for Algorithm (b), since D_i is obviously smaller than dC_i/dx; (2) Each P_i is computed in Algorithms (a) via one more division than by Algorithm (b).

Computation of GCD's often has the ultimate goal of cancelling the GCD and obtain the simpler cofactors of the corresponding input polynomials (as in the case of simplifying a rational number or rational function. Thus, it is often convenient as well as efficient to compute all three useful outputs through the call of one function. In the case of the Modular GCD Algorithm as stated and analyzed by Brown [BRO71], the cofactors are essentially by-products of the GCD computation, hence the cost of getting all three outputs is the same as that of getting the GCD alone. With such an algorithm in mind, we readily see that Algorithm (b) wastes no operations at all as opposed to Algorithm (a). Similarly, Algorithm (c) also utilizes all three outputs of the GCD computations.

Let the operation $(G,A^*,B^*) \leftarrow \gcd(A,B)$ be consisting of a call to a GCD computing function which also produces the corresponding cofactors and a parallel assignment of G to the GCD, A* to A/G, and B* to B/G. Then the above three algorithms can be restated in more concise forms as follows:

(a) (C_1,D_1,W) <-- gcd(P,dP/dx);
For i=1, step 1, until C_i = 1, do:
(C_{i+1},D_{i+1},W) <-- gcd(C_i,dC_i/dx);
P_i <-- D_i/D_{i+1}.

(b) (C_1,D_1,W) <-- gcd(P,dP/dx);
For i=1, step 1, until D_i=1, do:
(D_{i+1},C_{i+1},P_i) <-- gcd(C_i,D_i).

(c) (W,C_1,D_1) <-- gcd(P,dP/dx);
For i=1, step 1, until C_i=1, do:
(P_i,C_{i+1},D_{i+1}) <-- gcd(C_i,D_i-dC_i/dx).

With these reformulated algorithms, it is easier to see the extraneous efforts involved in Algorithm (a). "W" denotes unused output of GCD computations and Algorithm (a) has one such waste

at each iteration. In addition, there is also one extra division for each iteration of Algorithm (a).

Both Algorithms (b) and (c) have no wasted outputs from GCD computations within the "do-Loops". As a rough comparison between the two algorithms, we again compare the sizes (e.g., degrees) of the inputs to the GCD calculations. For Algorithm (b), the inputs are $C_i = P_{i+1}P_{i+2}^2 \cdots P_k^{k-1}$ and $D_i = P_i \cdots P_k$ while for Algorithm (c), they are $C_i = P_iP_{i+1} \cdots P_k$ and $D_i = P_i\left(\frac{dP_{i+1}}{dx} \cdots P_k + 2P_{i+1}\frac{dP_{i+2}}{dx} \cdots P_k + \cdots + (k-i)P_{i+1}\cdots\frac{dP_k}{dx}\right)$ where the degree of this D_i is one less than the degree of C_i. To give a more detailed comparison of these two algorithms, we will consider that the Modular GCD Algorithm [BRO71] is used which has a computing cost of

$$\ell^2(d+1)^v + \ell(d+1)^{v+1}$$

where ℓ is the maximum number of digits in the coefficient and d is the maximum degree in all v variables of the two input polynomials.

It is also necessary to make some assumptions on the sizes of the given polynomial P and the desired answers. Let $P = P_1P_2^2 \cdots P_k^k$ where $\deg(P_i) = d$, for $i = 1,2,\ldots,k$. Then $\deg(P) = dk(k+1)/2$. Assume also that N is the coefficient bound for, or the largest coefficient in P_i for all i. Then the length of N or the number of bits in N is $n = \log N$. For any j^{th} power of P_i, its size bounds in terms of coefficient length and degree are jn and jd, respectively.

For both Algorithms (b) and (c), C_i is the larger of the two inputs to the GCD computation at each iteration, so the total cost for the algorithms is

$$\text{Cost} = \sum_{i=1}^{k} [\text{length}(C_i)^2 (\deg(C_i) + 1)^v + \text{length}(C_i)(\deg(C_i) + 1)^{v+1}].$$

As we shall see, the univariate case will already show a difference in the computing costs of Algorithms (b) and (c). So let v = 1,

$$\text{Cost} \geq \sum_{i=1}^{k} \text{length}(C_i)^2\deg(C_i) + \sum_{i=1}^{k} \text{length}(C_i)\deg(C_i)^2.$$

The input polynomials to the GCD computations for the loop in Algorithm (b) are C_i and D_i. It is easy to see that C_i is usually the larger polynomial in terms of both degree, $\sum_{j=i+1}^{k} jd = d(k-i)(k+i+1)/2$, and coefficient length $\sum_{j=i+1}^{k} jn = n(k-i)(k+i+1)/2$. Using these bounds on sizes, we get

$$\text{Cost(b)} = \sum_{i=1}^{k} \frac{(n(k-i)(k+i+1))^2}{2} \cdot \frac{d(k-i)(k+i+1)}{2} + \sum_{i=1}^{k} \frac{nd^2(k-i)^3(k+i+1)^3}{8} = (n^2d + nd^2) \cdot \frac{k(k-1)(2k-1)(12k^4 - 24k^3 + 21k^2 - 9k+4)}{420}.$$

Correspondingly, the inputs to the GCD computations in Algorithm (c) are C_i and D_i where $\deg(D_i) = \deg(C_i) - 1$. Since $C_i = P_iP_{i+1} \cdots P_k$, $\deg(C_i) = (k-i+1)d$ and $\text{length}(C_i) = (k-i+1)n$. We then get

$$\text{Cost(c)} = \sum_{i=1}^{k} (n(k-i+1))^2 (d(k-i+1)) = \sum_{i=1}^{k} (n(k-i+1))(d(k-i+1))^2 = (n^2d+nd^2)\frac{k^2(k+1)^2}{4}.$$

Roughly speaking, cost (b) is approximately $O((2/35)k^7(n^2d + nd^2))$ as compared to cost(c) which is $O((1/4)k^4(n^2d + nd^2))$.

The computing cost for the Modular GCD Algorithm is slightly different in case the two input polynomials are relatively prime, i.e., GCD = 1. A formula of $\ell^2(d+1)^v+(d+1)^{v+1}$ is given by Brown [BRO71]. In view of that, we point out that while no GCD in the loop of Algorithm (b) is 1 until the last, as many GCD's as there are P_i's that are 1 can be equal to 1 in Algorithm (c). Thus we briefly show the difference in the computing costs of

these two algorithms for a special case where P_i's are equal 1 and have sizes (n,d) alternatingly. Let $P = P_1 P_i^2 \ldots P_k^k$ where deg $(P_i) = d$, for all $i = 2j$, $j = 1,2,\ldots,\ell$, $k = 2\ell$, and $P_i = 1$, for all $i = 2j-1$, $j = 1,2,\ldots,\ell$. Thus $\deg(P) = \sum_{j=1}^{\ell} 2jd = d\ell(\ell+1)$.

For Algorithm (b), again C_i is larger than D_i where

$$\deg(C_i) = \begin{cases} \sum_{j=\frac{i+1}{2}}^{\ell} (2j-1)d = (\ell^2 - (\frac{i-1}{2})^2)d & \text{for i odd} \\ \sum_{j=i/2}^{\ell} 2jd = (\ell+i/2)(\ell+i/2+1) & \text{for i even} \end{cases}$$

So

$$\text{Cost (b)} = (n^2d + nd^2)\left(\sum_{j=1}^{\ell}(\ell^2-(j-1)^2)^3 + \sum_{j=1}^{\ell}(\ell+j)^3(\ell+j+1)^3\right)$$

$$= (n^2d + nd^2)\,\frac{\ell(d+1)(186\ell^5 + 449\ell^4 + 357\ell^3 + 93\ell^2 + \ell - 1)}{10}$$

$$= O((93/640)\,k^7(n^2d + nd^2)).$$

For Algorithm (c), $\deg(C_i) = \deg(D_i) + 1$ or

$$\deg(C_i) = \begin{cases} (\ell - \frac{i+1}{2} + 1)\,d & \text{for i odd} \\ (\ell - \frac{i}{2} + 1)\,d & \text{for i even.} \end{cases}$$

So

$$\text{Cost(c)} = \sum_{j=1}^{\ell} n^2d(\ell-j+1)^3 + \sum_{j=1}^{\ell} d^2(\ell-j+1)^2 \quad (\text{GCD} = 1)$$

$$+ (n^2d+nd^2)\sum_{j=1}^{\ell}(\ell-j+1)^3 \quad (\text{GCD} \neq 1)$$

$$= (2n^2d+nd^2)\,\frac{\ell^2(\ell+1)^2}{4} + \frac{d^2\ell(\ell+1)(2\ell+1)}{6}$$

$$= O((1/32)k^4n^2d + (1/64)k^4nd^2 + (1/24)k^3d^2).$$

Note that due to all the rounding-off these comparisons of computing cost formulas do not show realistic differences until the parameter k becomes large. Some empirical testing bears this fact out such that the actual computing times for Algorithm (c), although always stays below that of Algorithm (b), do not show drastic differences for small values of k(up to k = 6).

4. Multivariate SQFR Algorithms

Yun proposed and implemented a square-free decomposition algorithm for multivariate polynomials [YUN74] [MAC73] which is based on a generalization of the Hensel's lifting technique called the Multivariate EZ Algorithm [YUN74], [M&Y73], [W&R75]. This SQFR algorithm, called EZSQFR Algorithm, parallels very much the outline of the EZGCD Algorithm of Moses and Yun [M&Y73]. It uses the technique of evaluation homomorphisms and the Hensel-type or p-adic constructions [YUN74] [YUN75] in much the same way as Brown's Modular GCD Algorithm uses homomorphisms and the Chinese Remainder Algorithm. We will not discuss these techniques in detail for the lack of space, but only refer interested readers to the above references.

Before presenting the multivariate EZSQFR Algorithm we will first discuss a useful, time-saving device for the computation of square-free decompositions, due to a suggestion by P.Wang. A given multivariate polynomial could already be square-free with respect to a particular variable. If this fact cam be detected, at a relatively small cost, before the entire machinery of the square-free algorithm begins to work, there can be big savings in computing time. There is such a time saving test which we shall call <u>fail-safe square-free test</u> (f.s.s.f.). This test consists of (a) evaluating the polynomial at random by chosen "valid" ([YUN74] or see † below) points. (b)

computing the GCD of the resulting univariate polynomial and its derivative w.r.t. the main variable. (c) checking to see if the GCD is an integer or not, if so, then the original polynomial is square-free. The validity of this test is shown by the following lemma:

Lemma 2: Let $b = (b_1,\ldots,b_v)$ be an arbitrary valid evaluation for $P(x,y_1,\ldots,y_v)$ in $Z[x,y_1,\ldots,y_v]$. If $P_b(x) = P(x,b_1,\ldots,b_v)$ is square-free in $Z(x)$ then P is itself square-free.

Proof: If P is not square-free, $P = QR^2$, then via the evaluation homomorphism $P_b = Q_b R_b^2$ which cannot be square-free. //

The usefulness of this test comes from the next lemma, which is also stated and proved by Wang and Rothschild [W&R75]:

Lemma 3: If P is a square-free multivariate polynomial in $Z[x,y_1,\ldots,y_v]$, then a set of integers $b = (b_1,\ldots,b_v)$ can be chosen so that be is a valid evaluation for P(i.e., $\dagger$ $\deg(P)=\deg(P_b(x))$ and $P_b(x)$ is also square-free.

Proof: Let $P = P_1P_2 \ldots P_k$ be the factorization of P into irreducible factors where $\gcd(P_i,P_j) = 1$ for all i and j such that $i \neq j$. $P_b(x)$ is square-free if and only if $P_{i_b}(x)$ is square-free for all i and $\gcd(P_{i_b}(x),P_{i_b}(x))$ = constant, for all $i \neq j$, which is equivalent to $\mathrm{reslt}(P_{i_b}(x),P_{j_b}(x))\neq 0$, for all i,j, $i\neq j$, where "reslt" denotes the resultant w.r.t. x[VDW49]. Let $R(x,y_1,\ldots,y_v) = \prod_i \mathrm{reslt}(P_i,dP_i/dx) \prod_{i<j} \mathrm{reslt}(P_i,P_j)$. Then $R \neq 0$ since P is square-free and $P_b(x)$ is square-free if and only if $R_b(x)\neq 0$. Now it suffices to note that there are only finitely many integral values for $b_1,\ldots,b_v$ such that $R_b(x) = 0$ or $(\mathrm{lc}(P))_b = 0$. //

The finiteness of integral roots for $R(x,y_1,\ldots,y_v)$ has the further implication that out of the infinite possibilities of integral values for each b, the probability for choosing a valid but unlucky set b, such that $P_b(x)$ becomes not square-free when P is, cannot be too large. We will apply this test at the beginning of the EZSQFR Algorithm with at most two valid evaluations. If the test fails, we simply assume no information was gained and continue with the rest of the EZSQFR Algorithm. However, we point out that not all the work done for the test was wasted in this case, since the computations done for the f.s.s.f. test is useful for the EZGCD Algorithm when it is used to compute the square-free part of the given polynomial.

Next, we will prove a lemma which will eliminate the need for using the special case method of the EZGCD Algorithm [M&Y73] when it is invoked to compute N = gcd(P, dP/dx) at the second step of the EZSQFR Algorithm.

Lemma 4: Let P be primitive in $(Z[y_1,\ldots,y_v])[x]$ and R = gcd(P, dP/dx), then gcd(R, (dP/dx)/R) = 1.

Proof: Let $P = P_1P_2^2 \ldots P_k^k$ and Q = dP/dx. By Theorem 1, $R = P_2^1 P_3^2 \ldots P_k^{k-1}$. Assume $D = \gcd(R, Q/R) \neq 1$, then $Q = R\,D\,Q''$ for some Q". Since D divides R, there must be an irreducible factor $C \neq 1$ of some P_i, $i \geq 2$. But this C clearly also divides P/R so that $P = R\,C\,P''$ for some P". Thus, R C divides both P and Q, contradicting to R being the GCD. Therefore, gcd(R, Q/R) must be 1. //

This lemma and the finiteness of the number of unlucky evaluations imply that the polynomial dP/dx can always be used for the application of the p-adic construction. Also, the special case

method of the EZGCD Algorithm can be completely avoided for this square-free decomposition algorithm. We are ready now to describe the EZSQFR Algorithm in detail. However, we will not be so careful as to indicate, for example, how the computations for f.s.s.f test is used in the EZGCD Algorithm. If the given polynomial is not primitive, then clearly we can work on its content and primitive part separately. Because of the fact that square-free decompositions are main variable dependent and that for some uses it is not necessary to square-free decompose the content, we will assume primitive inputs to the EZSQFR Algorithm.

Algorithm (d): (EZSQFR)

Input: A primitive multivariate polynomial $P(x,y_1,\ldots,y_v)$ in $Z[x,y_1,\ldots,y_v]$ and the main variables, x.

Output: A list of polynomials in $Z[x,y_1,\ldots,y_v]$, $P_1,P_2,\ldots,P_k$, such that $P = P_1P_2^2 \ldots P_k^k$ for some $k \geq 1$ where $\gcd(P_i,dP_i/dx) = 1$, $\gcd(P_i,P_j) = 1$ for $i \neq j$, and $P_k \neq 1$.

(1) Invoke the f.s.s.f. test twice with random valid evaluations. If the test is successful, then return P, otherwise continue.

(2) Apply the EZGCD Algorithm on P and dP/dx to obtain $N = \gcd(P,dP/dx)$ and $L = P/N$. Let b be the valid lucky evaluation for P and p be the lucky prime for $P_b(x)$ used in the EZGCD Algorithm above.

(3) Set $L_b \leftarrow L(x,b_1,\ldots,b_v)$, $N_b \leftarrow N(x,b_1,\ldots,b_v)$, and $i \leftarrow 1$.

(4) Apply UNIGCO Algorithm (or some other univariate GCD Algorithm which also computes cofactors) on L_b and N_b to obtain $H_o \leftarrow \gcd(L_b, N_b)$, $G_o \leftarrow L_b/H_o$, and $N_b \leftarrow N_b/H_o$. Set $L_b \leftarrow H_o$.

(5) If $G_o = 1$, then set $P_i \leftarrow 1$. Otherwise, apply the Multivariate EZ Algorithm on L, G_o, and H_o to get multivariate polynomials G and H such that $L = G\,H$ over Z. Set $P_i \leftarrow G$, $L \leftarrow H$, and $i \leftarrow i + 1$.

(6) If $N_b = 1$, then go to (4). Otherwise, set $k \leftarrow 1$, $P_k \leftarrow L$, and return $P_1, P_2,\ldots, P_k$.

Remark: Note that even the evaluation of multivariate polynomial L and N in Step (3) need only be done once. The later values for L_b and N_b are simply updated with known quantities. Step (4) involves only one univariate GCD computation. For the application of the Multivariate EZ Algorithm in Step (5), the required lucky evaluation b and prime p is again provided by the EZGCD Algorithm in Step (2). Thus the results of that computation are certain to be correct over Z.

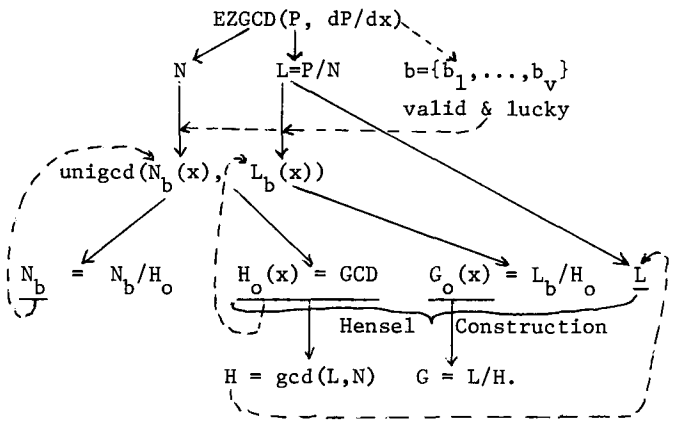
The EZSQFR Algorithm takes great advantages of computations already performed to achieve added efficiencies. By performing Hensel constructions instead of full-fledged multivariate GCD computations, the gains in efficiency are clear. Even the preparations for these applications of the Multivariate EZ Algorithm are very simple. They involve essentially only one univariate GCD computation as preparation for each Hensel construction, hence for each P_i. That is much less costly than what EZGCD Algorithm goes through for preparing Hensel construction. All the choosing of lucky evaluations and primes, testing for various conditions etc., are eliminated because the evaluation values and the prime are known to be lucky after the first call to EZGCD.

Whenever $P_i = 1$ for some i (for example, $P = U\,V^3$, then $P_1 = U, P_2 = 1, P_3 = V$), the computational process is even simpler - only one uni-

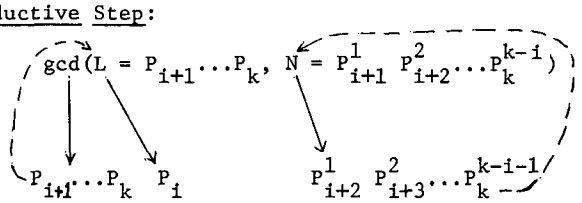
variate GCD computation is necessary and no multivariate operation is required at all. Compared to Algorithm (b), we would find in this case, $D_{i+1} = \gcd(C_i, D_i) = D_i$, so that $P_i = D_i/D_{i+1} = 1$, i.e. D_i divides C_i. Even so, there is still at least one multivariate division involved, which is more time consuming than one univariate GCD computation in most cases.

Algorithm (d), EZSQFR, as presented in detail above clearly is closely related to Algorithm (b). For the GCD computations in the loop, both algorithms use as inputs the remaining square-free part and the non-square-free part, i.e., Algorithm (b) uses $D_i = P_i \ldots P_k$ and $C_i = P_{i+1}P_{i+2}^2 \ldots P_k^{k-i}$ whereas Algorithm (d) uses the univariate images, $L_b(x)$ and $N_b(x)$, of these. To make the flow of ideas clearer, we present a schematic diagram of the EZSQFR Algorithm (d), together with a short concise version of the Algorithm.

Schematic Diagram of EZSQFR



Inductive Step:



Algorithm (d): EZSQFR (applying the Hensel Construction)

(i) $(N, L, W) \leftarrow$ EZGCD $(P, dP/dx)$.

(ii) Use the valid and lucky evaluation $b = \{b_1, \ldots, b_v\}$ for P from above EZGCD computation to compute $L_b(x) = L(x, b_1, \ldots, b_v)$ and $N_b(x) = N(x, b_1, \ldots, b_v)$. Set $i \leftarrow 1$.

(iii) Loop: $(L_b, G_0, N_b) \leftarrow$ unigcd $(L_b(x), N_b(x))$ (univariate GCD computation). If $G_0 = 1$, then set $P_i \leftarrow 1$, otherwise apply the Hensel Construction on L, G_0, $H_0 \leftarrow L_b$ to get G and H such that $L = GH$. Set $P_i \leftarrow G$, $L \leftarrow H$, and $i \leftarrow i+1$. If $N_b(x) \neq 1$, go Loop, else set $P_i \leftarrow L$ and return $P_1, P_2, \ldots, P_k$ where $k = i$.

Corresponding to Algorithm (c), it is obviously also possible to extend and generalize that to the multivariate case and apply the Hensel or Multivariate EZ Algorithm. We present such a version only in short form as Algorithm (e) and give a schematic diagram as follows

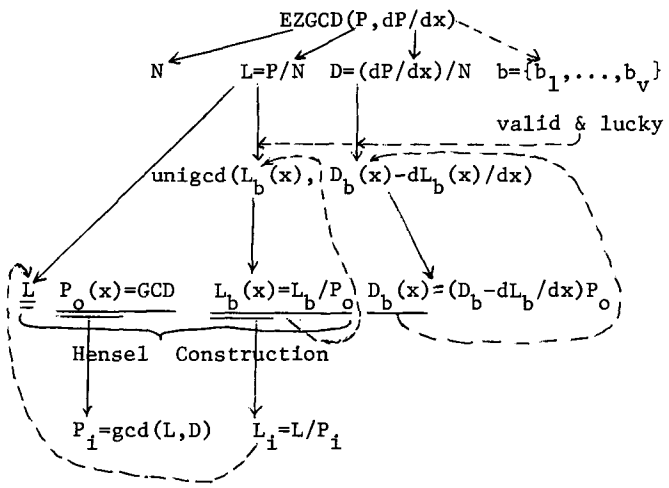
Algorithm (e):

(i) $(N, L, D) \leftarrow$ EZGCD $(P, dP/dx)$

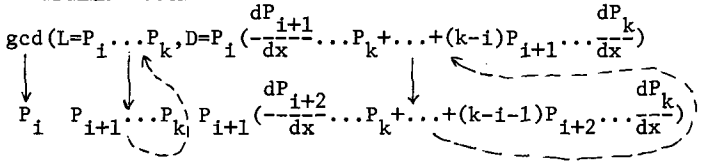
(ii) Use the valid and lucky evaluation points $b = \{b_1, \ldots, b_v\}$ for P from the above EZGCD computation to get $L_b(x) = L(x, b_1, \ldots, b_v)$ and $D_b(x) = D(x, b_1, \ldots, b_v)$. Set $i \leftarrow 1$.

(iii) Loop: $(P_0, L_b, D_b) \leftarrow$ unigcd $(L_b(x), D_b(x) - dL_b(x)/dx)$. If $P_0 = 1$, then set $P_i \leftarrow 1$, otherwise apply the Hensel Construction on L, P_0, L_b to get P_i and $L_i = P_i L_i$. Set $L \leftarrow L_i$ and $i \leftarrow i+1$. If $L \neq 1$, go Loop, else set $P_i \leftarrow L$ and return $P_1, P_2, \ldots, P_k$ where $k \leftarrow i$.

Schematic Diagram of Algorithm(e)



Inductive Step:



In summary, both Algorithm (d) and (e), while being generalizations of Algorithms (b) and (c), respectively, show that the cost for computing square-free decompositions of multivariate polynomials is essentially the same as the cost of computing GCD's. To see that, we need only realize that all the Hensel Construction applications in the loops of both algorithms amount to the work of lifting the relatively prime factors of $L = P_1P_2 \ldots P_k$. However, that effort is dominated by the cost of lifting L and D = (dP/dx)/L from dP/dx in the Hensel Construction stage of the EZGCD computation. All the other work in both algorithms is comparatively negligible. Therefore, the cost of EZGCD(P, dP/dx) is the dominant cost for Algorithms (d) and (e).

5. Conclusions

In this paper, we presented three algorithms for directly computing the square-free decomposition of polynomials by GCD calculations. Two of them, Algorithm (a) and (b), have been known. They are included here for completeness and comparison. Algorithm (c) involves surprisingly simple ideas, yet it quite obviously constitutes an improvement over the other two. Further improvements are realized as we consider the advantages of Modular GCD Algorithms. Several comments by Brown [BRO71] on the efficiency of the modular approach to GCD computations are actually put in practice here: (1) Modular GCD Algorithm computes the GCD and the corresponding cofactors essentially at no extra cost; (2) In terms of dense polynomial arithmetic, the cost of Modular GCD Algorithm is comparable to that of one division of the same polynomial inputs; (3) Modular approach to GCD computation achieves added efficiency when the input polynomials are relatively prime. The advantages resulting from all these points on algorithm efficiency are taken by Algorithm (c), as demonstrated by our analysis and comparison of these three algorithms.

Without losing the advantages of the modular approach, the Hensel or p-adic construction is brought into the computation of SQFR decompositions, especially in the multivariate case. Algorithms (b) and (c) are extended respectively to Algorithms (d) (i.e. EZSQFR) and (e). While preserving their original efficiencies, they showed the new feature that the cost of computing the SQFR decomposition of a polynomial is comparable (related by a constant factor) to that of computing the GCD of the same polynomial and its derivative.

Based on that conclusion, we feel the SQFR decomposition can be effectively used as an improvement to the factored representation of rational functions as proposed by Brown [BRO74] and Hall [HAL74]. Since their method obtains factors by

GCD computations while avoiding the expensive process of complete factorization, the SQFR factored representation will provide more factors also at the expense of GCD computations. Furthermore the SQFR representation of polynomials is canonical which is not the case for the partially factored numerator and denominator of rational functions in their scheme. Therefore, the possibility of using the square-free canonical representation for polynomials in some future algebraic systems (at a cost not much more than a GCD computation) is enhanced and suggested.

REFERENCES

[BRO71] Brown, W.S., "On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors", JACM, Vol. 18, No. 4, October 1971, pp. 478-504.

[BRO74] Brown, W.S., "On Computing with Factored Rational Expressions", Proceedings of EUROSAM '74, (ACM SIGSAM Bulletin No. 31), pp. 27-34.

[C&F76] Caviness, B.F., and Fateman, R.J., "Simplication of Radical Expressions", these proceedings.

[FAT72] Fateman, R.J., "Essays in Algebraic Simplification", Ph.D Thesis, Harvard, also MAC TR-95, MIT, April 1972.

[GJY75] Griesmer, J.H., Jenks, R.D., and Yun, D.Y.Y., "SCRATCHPAD User's Manual", IBM T.J. Watson Research Center Report - RA70, June 1975.

[HAL74] Hall, A.D., "Factored Rational Expressions in ALTRAN", Proceeding of EUROSAM '74, pp. 35-45.

[HEN13] Hensel, K., Zahlentheorie, Goschen, Berlin and Leipzig, 1913.

[HOR69] Horowitz, E., "Algorithms for Symbolic Integration of Rational Functions", Ph.D. Thesis, Computer Science Dept., University of Wisconsin, 1969.

[HOR71] Horowitz, E., "Algorithms for Partial Fraction Decomposition and Rational Function Integration", [PET71], pp. 441-457.

[KNU69] Knuth, D.E., The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, Addison Wesley, Reading, Mass. 1969.

[MAC73] MACSYMA User's Manual, Bogen, R., et al., Project MAC, M.I.T., Cambridge, Mass.,1973.

[MOS67] Moses, J., "Symbolic Integration", MAC-TR-47, Project MAC, M.I.T., Dec., 1967.

[MOS71] Moses, J., "Symbolic Integration: The Stormy Decade", [PET71], March 1971, pp. 427-440.

[MUS71] Musser, D.R., "Algorithms for Polynomial Factorization", Ph.D. Thesis, C.S. Dept., Univ. of Wisconsin, August 1971.

[M&Y73] Moses, J. and Yun, D.Y.Y., "The EZGCD Algorithm", Proc. of ACM Annual Conference, Aug. 1973, Atlanta, pp. 159-166.

[M&Y74] Miola, A. and Yun, D.Y.Y., "The Computational Aspects of Hensel-type Univariate Polynomial Greatest Common Divisor Algorithms", Proc. EUROSAM '74, Aug. 1974, pp. 46-54.

[PET71] Proc. of the Second Symposium on Symbolic and Algebraic Manipulation, Petrick, S.R., ed., ACM, March 1971.

[RIS69] Risch, R., "The Problem of Integration in Finite Terms", Trans. AMS, Vol. 139, May 1969, pp. 167-189.

[TOB67] Tobey, R.G., "Algorithms for Antidifferentiation of Rational Functions", Ph.D. Thesis, Harvard, 1967.

[W&R75] Wang, P.S. and Rothschild, L.P., "Factoring Multivariate Polynomials over the Integers", Math. of Comp. Vol. 29, No. 131, July 1975, pp. 935-950.

[YUN74] Yun, D.Y.Y., "The Hensel Lemma in Algebraic Manipulation", Ph.D. Thesis, Dept. of Math. MAC-TR-138, M.I.T., Nov. 1974.

[YUN75] Yun, D.Y.Y., "Hensel Meets Newton -- Algebraic Construction in an Analytic Setting", Analytic Computational Complexity, Proc. of CMU Symposium, Traub, J., ed., Academic Press, April 1975.

[YUN76] Yun, D.Y.Y., "P-adic Constructions and its Applications in Algebraic Manipulation", these proceedings.