Improving the Multiprecision Euclidean Algorithm *

Tudor Jebelean
RISC-LINZ

Research Institute for Symbolic Computation
Johannes Kepler University, A4040 Linz, Austria (Europe)
Tel: +43 (7236) 3231-49, -20 Fax: +43 (7236) 3231-30
e-mail: tjebelea@risc.uni-linz.ac.at

June 10, 1993

Abstract

We improve the implementation of Lehmer's algorithm for multiprecision integer GCD computation by partial cosequence computation on pairs of double digits, enhanced condition for exiting the partial cosequence computation, and approximative GCD computation. The combined effect of these improvements is an experimentally measured speed-up by a factor of 2 over the currently used implementation.

Introduction

The speed of multiprecision arithmetic has a decisive influence on the speed of computer algebra systems. By performing a series of measurements upon typical algebraic algorithms [Buchberger, Jebelean 92], we found out that most of the computation time is spent in integer/rational operations, and this proportion increases when the length of the input coefficients increases. Also, the absolute total time increases dramatically, exclusively on behalf of integer/rational operations. For example, in Gröbner Bases computation [Buchberger 85], when increasing the coefficient length from 1 to 10 decimal digits, the proportion of rational operations grows from 62% to 98%, and the total computation time grows by a factor of 25. Also, we noticed that GCD computation is the most time consuming operation on long integers (25% to 70% of the total computing time).

For long integers which occur in typical applications (up to 100 computer words), it is generally accepted that the best GCD scheme is Lehmer's multiprecision Euclidean algorithm (see [Lehmer 38], [Knuth 81], [Collins 80]). FFT based algorithms (see [Schönhage 71], [Moenck 73]), although having a better theoretical complexity, are not efficient for integers in this range.

We present here three ways of improving Lehmer's algorithm:

- partial cosequence computation using pairs of double digits,
- a better method for detecting the end of the partial cosequence computation,

*Acknowledgements:
Austrian Ministry for Science and Research, project 613.523/3-27a/89 (Gröbner Bases) and doctoral scholarship;
POSSO project(Polynomial Systems Solving – ESPRIT III BRA 6846);

- approximative GCD computation.

The combined effect of these improvements leads to a speed-up by a factor of 2 over the (currently used) straight forward implementation of Lehmer's GCD algorithm.

1 The multiprecision Euclidean algorithm

We present here the Euclidean algorithm and the multiprecision version of it [Lehmer 38] and we notice two useful properties of the cosequences. More details can be found in [Knuth 81] and [Collins 80]. We adopt here (with small modifications) the notations used in [Collins 80].

Let $A > B > 0$ be integers.

The Euclidean algorithm consists in computing the *remainder sequence* of (A, B): $(A_0, A_1, \ldots, A_n, A_{n+1})$ defined by the relations

$$\begin{aligned} A_0 = A, \quad A_1 = B,& \\ A_{i+2} = (A_i)_{\mathrm{mod}\ A_{i+1}},& \\ A_{n+1} = 0.& \end{aligned} \tag{1}$$

It is well known that

$$A_n = GCD(A, B).$$

The extended Euclidean algorithm consists in computing additionally the *quotient sequence* $(Q_1, \ldots, Q_n)$ defined by

$$Q_{i+1} = \lfloor A_i / A_{i+1} \rfloor,$$

and the *first* and *second cosequences* $(U_0, U_1, \ldots, U_{n+1})$ and $(V_0, V_1, \ldots, V_{n+1})$:

$$\begin{aligned} (U_0, V_0) &= (0, 1), \\ (U_1, V_1) &= (1, 0), \\ (U_{i+2}, V_{i+2}) &= (U_i, V_i) - Q_{i+1}(U_{i+1}, V_{i+1}). \end{aligned}$$

Then the following hold:

$$A_{i+2} = A_i - Q_{i+1} A_{i+1}, \tag{2}$$

$$A_i = AU_i + BV_i. \tag{3}$$

It is useful to note that the signs of the elements of each cosequence alternate. This can be seen if we consider the sequences $(X_0, X_1, \ldots, X_{n+1})$ and $(Y_0, Y_1, \ldots, Y_{n+1})$ defined by

$$\begin{aligned} (X_0, Y_0) &= (0, 1), \\ (X_1, Y_1) &= (1, 0), \\ (X_{i+2}, Y_{i+2}) &= (X_i, Y_i) + Q_{i+1}(X_{i+1}, Y_{i+1}). \end{aligned} \tag{4}$$

Then

$$\begin{aligned} U_i &= (-1)^i X_i = (-1)^i |U_i|, \\ V_i &= (-1)^{i+1} Y_i = (-1)^{i+1} |V_i|. \end{aligned} \tag{5}$$

Indeed, the relations are immediate for $i = 0, 1$ and the induction step is:

$$\begin{aligned} U_{i+2} &= U_i - Q_{i+1} U_{i+1} \\ &= (-1)^i X_i - Q_{i+1} (-1)^{i+1} X_{i+1} \\ &= (-1)^i (X_i + Q_{i+1} X_{i+1}) = (-1)^{i+2} X_{i+2}, \end{aligned}$$

and similar for V_{i+2}. The second parts of equalities (5) are a consequence of $X_i, Y_i \geq 0$. Note also that (4) and (5) implies

$$(|U_{i+2}|, |V_{i+2}|) = (|U_i|, |V_i|) + Q_{i+1}(|U_{i+1}|, |V_{i+1}|). \tag{6}$$

These relations are useful for implementing the algorithm when the upper value of U_i, V_i is the maximum value which can be contained in a computer word (e.g. $2^{32} - 1$), because then the signs are difficult to handle, since the sign-bit cannot be used.

Another useful relation is

$$|V_i| \;\geq\; Q_1|U_i|, \quad \text{for all } i \geq 1. \tag{7}$$

Indeed, the relation can be verified directly for $i = 1, 2$ and the induction step is:

$$|V_{i+2}| = Y_i + Q_{i+1}Y_{i+1} \geq Q_1X_i + Q_{i+1}Q_1X_{i+1} = Q_1|U_{i+2}|.$$

When A and B are multiprecision integers (several computer words are needed for storing the values), the divisions in (1) are quite time consuming. [Lehmer 38] noticed that several steps of the Euclidean algorithm can be simulated by using only simple precision divisions. The idea is to apply the extended algorithm to

$$a = \lfloor a/2^h \rfloor, \quad b = \lfloor B/2^h \rfloor,$$

where $h \geq 0$ is chosen such that $a > b > 0$ are simple precision.

Then one gets the remainder sequence $(a_0, a_1, ..., a_k, a_{k+1})$, the quotient sequence $(q_1, \ldots, q_k)$ and the cosequences $(u_0, u_1, \ldots, u_k, u_{k+1})$ and $(v_0, v_1, \ldots, v_k, v_{k+1})$, for some $k \geq 0$ such that

$$q_i = Q_i, \quad \text{for all } i \leq k. \tag{8}$$

This process is called *digit partial cosequence calculation.* When (8) is true, we say *the k-length quotient sequences of (a,b) and (A,B) match.* Then also

$$(u_i, v_i) = (U_i, V_i), \quad \text{for all } i \leq k+1,$$

hence by (3):

$$\begin{aligned} A_k &= u_kA + v_kB, \\ A_{k+1} &= u_{k+1}A + v_{k+1}B, \end{aligned} \tag{9}$$

and the process can be reiterated with the most significant digits of A_k, A_{k+1}.

However, the condition (8) cannot be tested directly, since Q_i are not known, hence one must have a condition upon a_i, q_i, u_i, v_i which ensures (8).

Lehmer's original algorithm computed the quotient sequences of $(a, b{+}1)$ and $(a{+}1, b)$ and compared the quotients at each step.

[Collins 80] developed a better condition which needs only the computation of the sequences (q_i), (a_i) and (v_i). That is

$$a_{i+1} \geq |v_{i+1}| \;\text{ and }\; a_{i+1} - a_i \geq |v_{i+1} - v_i|, \quad \text{for all } i \leq k. \tag{10}$$

This condition has the advantage that only one quotient has to be computed, and only one of the cosequences. As noted by G. H. Bradley (acc. to [Knuth 81]), u_k, u_{k+1} can be found at the end of partial cosequence computation, by using

$$\begin{aligned} u_ka + v_kb &= a_k, \\ u_{k+1}a + v_{k+1}b &= a_{k+1}. \end{aligned} \tag{11}$$

As a comparison base for our improvements, we implemented this version of the multiprecision Euclidean algorithm using the GNU multiprecision library [Granlund 91] and GNU optimizing compiler on a Digital DECstation 5000. The benchmarks for integers with lengths varying from 5 to 100 digits (32 bit words) are shown in Fig.1(columns S). 1000 random integers were tested for each length. These experimental settings will remain unchanged for all the benchmarks we will present in the rest of this paper.

Length	Steps per digit			Divisions per step			Time (ms)		
of operands	S	D	S/D%	S	D	S/D%	S	D	S/D%
5	1.52	0.80	190.0	7.74	17.23	44.9	0.73	1.90	38.4
10	2.00	0.91	219.8	7.63	17.04	44.8	2.10	4.35	48.3
15	2.16	0.99	218.2	7.60	16.97	44.8	3.82	7.30	52.3
20	2.24	1.01	221.8	7.56	16.91	44.7	5.87	10.13	57.9
25	2.28	1.03	221.4	7.58	16.94	44.7	8.25	13.30	62.0
30	2.31	1.04	222.1	7.57	16.94	44.7	11.02	16.63	66.3
35	2.34	1.05	222.9	7.57	16.92	44.7	14.10	20.00	70.5
40	2.35	1.06	221.7	7.57	16.93	44.7	17.52	23.58	74.3
45	2.37	1.07	221.5	7.57	16.91	44.8	21.32	27.60	77.2
50	2.38	1.07	222.4	7.57	16.92	44.7	25.35	31.32	80.9
55	2.38	1.07	222.4	7.57	16.93	44.7	29.85	35.38	84.4
60	2.39	1.07	223.4	7.57	16.93	44.7	34.67	39.52	87.7
65	2.40	1.08	222.2	7.58	16.93	44.8	39.87	43.85	90.9
70	2.40	1.08	222.2	7.56	16.91	44.7	45.28	48.32	93.7
75	2.41	1.08	223.1	7.57	16.93	44.7	51.17	53.03	96.5
80	2.41	1.08	223.1	7.57	16.93	44.7	57.35	57.88	99.1
85	2.42	1.08	224.1	7.57	16.93	44.7	63.85	62.87	101.6
90	2.42	1.08	224.1	7.58	16.94	44.7	70.75	67.98	104.1
95	2.42	1.09	222.0	7.57	16.93	44.7	78.02	73.33	106.4
100	2.42	1.09	222.0	7.56	16.91	44.7	85.82	78.67	109.1

Figure 1: Benchmarks of GCD computation using simple digit DPCC (columns S) vs. double digit DPCC2 (columns D).

2 The double digit algorithm

Recovering A_k, A_{k+1} by (9) involves 4 multiplications of a single digit number by a multidigit number, and this is the most time consuming part of the whole computation. Experimentally, for 32 bit digits, one notices that the final cofactors are usually shorter than 16 bits. Hence, if digit partial cosequences computation (DPCC) would be performed on double digits, then the recovering step would require the same computational effort, but will occur (roughly) two times less frequently. This is the main idea of the improvement of Lehmer's algorithm which we will discuss in this section.

A straight forward implementation of this idea is not very efficient (see Fig.1): the experimental speed up ranges from 38% (5 words operands) to 109% (100 words operands), hence is rather a slow down. However, the average values in the table show that, indeed, the number of divisions which are simulated within each step increases more than twice, which leads to a decrease of the number of steps by a factor of 2.2. The lack of speed is due to the fact that the partial cosequence computation is much slower, because of the double-digits operations involved. Therefore we will concentrate in the sequel on improving the partial cosequence computation for double words (DPCC2). Few simple improvements lead to a speed up of almost 2:

- Some double-digit operations can be reduced to simple digit operations by using

$$|u_i| \leq |u_{i+1}| \leq |v_{i+1}|, \quad |v_i| \leq |v_{i+1}|,$$

  which are a consequence of (6) and (7). Therefore, only v_{i+1} has to be computed with two digits. As long as its higher digit is null, the other cofactors are also small. The speed up ranges now from 43% to 117%.

- Digit partial cosequence computation for simple words (DPCC) can be used at the beginning of DPCC2. The speed up grows to 58% – 140%.
- Since only the second cosequence (v_i) is needed for testing the exit condition (10), the cofactors u_k, u_{k+1} can be computed at the end, using (11). However, our experiments show no improvement by this method: the speed up decreases to 53% – 133%.
- Because the quotients are usually small (see [Knuth 81]), simulating division by repeated subtractions is also a source of speed up. The idea is to repeat q_{small} times the subtraction cycle, and if the quotient is not found by then, to use division. Our experiment show that $q_{\text{small}} = 32$ is a good threshold value. The increase of the speed up is surprisingly high: we get 101% to 180%.

The experimental results for the above variants of the algorithm are presented in Fig. 2, and we keep the best variant as "version 1", which will be further improved in the sequel.

3 On the condition needed in digit partial cosequence computation

We develop here an "exact" condition for (8), that is, a condition which precisely indicates which is the highest k for which (8) is satisfied. Also we develop an "approximative" condition, which is weaker, but easier to test. We show then how to combine these two conditions, such that both efficiency and exactness are preserved.

3.1 An exact condition

Theorem 1. Let $a > b > 0$ be integers. Then

the k-length quotient sequences of (a, b) and $(2^h b + A', 2^h a + B')$ match for any $h > 0$ and any $A', B' < 2^h$

if and only if

$$\begin{gathered} a_{i+1} \geq -u_{i+1} \quad \text{and} \quad a_i - a_{i+1} \geq v_{i+1} - v_i \quad \text{and} \quad i \text{ even} \\ \text{or} \\ a_{i+1} \geq -v_{i+1} \quad \text{and} \quad a_i - a_{i+1} \geq u_{i+1} - u_i \quad \text{and} \quad i \text{ odd} \\ \text{for all } i \leq k. \end{gathered}$$

Proof. We keep all the notations as previously introduced. Let us also denote by $(A_0, A_1, \ldots, A_{k+1})$ the elements of the remainder sequence of

$$(A, B) = (2^h a + A', \quad 2^h b + B').$$

Then

$$A_i = u_i(2^h a + A') + v_i(2^h b + B') = 2^h(u_i a + v_i b) + u_i A' + v_i B' = 2^h a_i + u_i A' + v_i B'.$$

Applying the relation

$$q = \lfloor M/N \rfloor \ \text{ iff } \ 0 \leq M - qN < N \quad (M, N, q \text{ positive integers})$$

to

$$q_i = \left\lfloor \frac{A_{i-1}}{A_i} \right\rfloor = \left\lfloor \frac{2^h a_{i-1} + u_{i-1}A' + v_{i-1}B'}{2^h a_i + u_i A' + v_i B'} \right\rfloor = \left\lfloor \frac{a_{i-1} + u_{i-1}\frac{A'}{2^h} + v_{i-1}\frac{B'}{2^h}}{a_i + u_i\frac{A'}{2^h} + v_i\frac{B'}{2^h}} \right\rfloor,$$

one gets

$$0 \ \leq \ a_{i-1} + u_{i-1}\frac{A'}{2^h} + v_{i-1}\frac{B'}{2^h} - q_i\left(a_i + u_i\frac{A'}{2^h} + v_i\frac{B'}{2^h}\right) \ < \ a_i + u_i\frac{A'}{2^h} + v_i\frac{B'}{2^h}, \quad \forall h, A', B'. \tag{12}$$

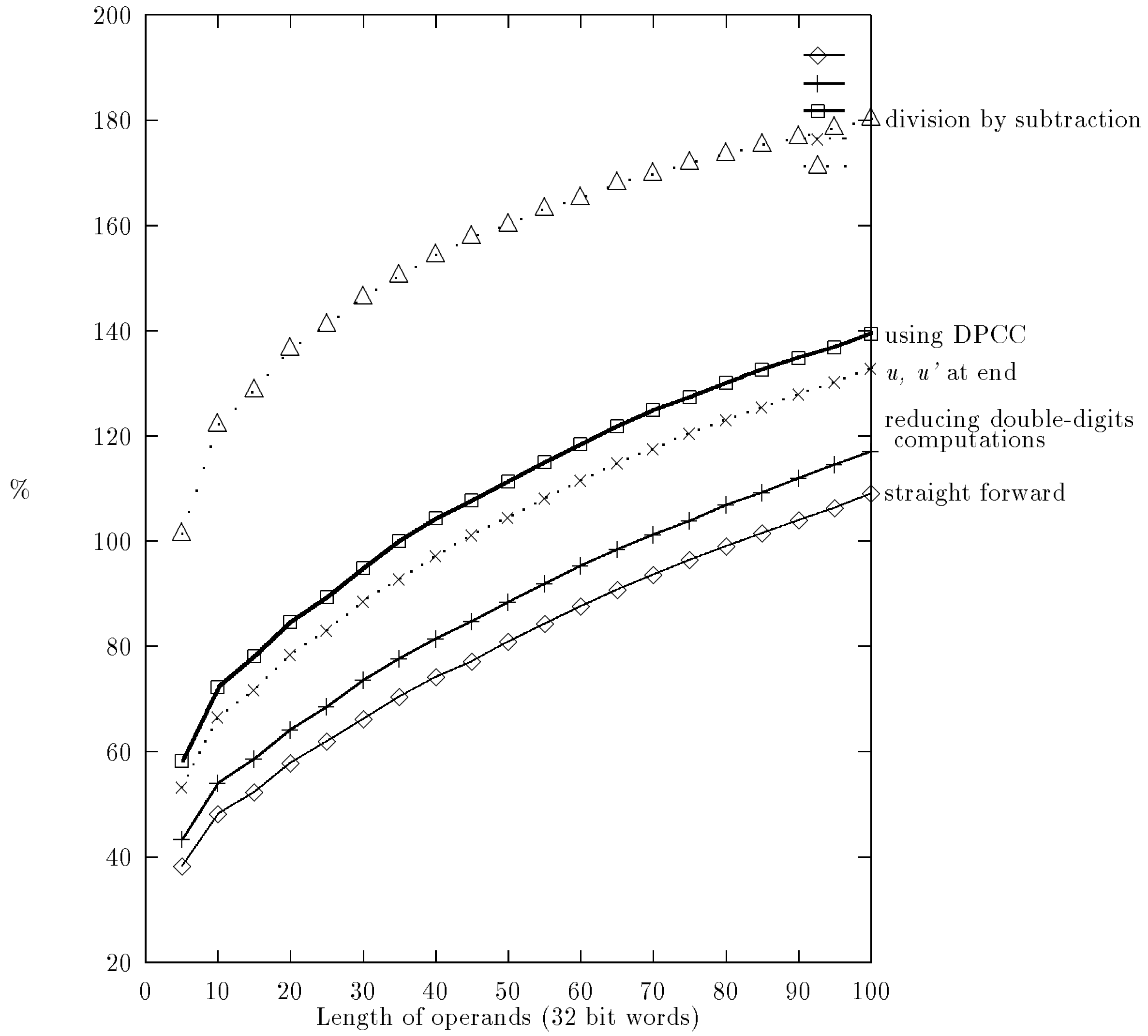

Figure 2: Speed up in % for version 1.

Using

$$u_{i+1} = u_{i-1} - q_i u_i,$$

$$v_{i+1} = v_{i-1} - q_i v_i,$$

one notes that (12) is equivalent to the conjunction of

$$0 \le \min\left\{ a_{i+1} + u_{i+1}\frac{A'}{2^h} + v_{i+1}\frac{B'}{2^h} \;\middle|\; 0 < h,\ 0 \le A', B' < 2^h \right\},$$

$$\max\left\{ a_{i+1} - a_i + (u_{i+1} - u_i)\frac{A'}{2^h} + (v_{i+1} - v_i)\frac{B'}{2^h} \;\middle|\; 0 < h,\ 0 \le A', B' < 2^h \right\} \le 0,$$

The reason why the second inequality $<$ can be replaced by $\le$ will become clear in the sequel.

Suppose i is even. Then

$$u_{i+1}, v_i \le 0, \quad u_i, v_{i+1} \ge 0,$$

hence

$$\begin{aligned} &\min\left\{ a_{i+1} + u_{i+1}\frac{A'}{2^h} + v_{i+1}\frac{B'}{2^h} \right\}, \\ &= a_{i+1} + u_{i+1}\max\frac{A'}{2^h} + v_{i+1}\min\frac{B'}{2^h} \\ &= a_{i+1} + u_{i+1} * 1 + v_{i+1} * 0 \\ &= a_{i+1} + u_{i+1}. \end{aligned}$$

Also

$$\begin{aligned} &\max\left\{ a_{i+1} - a_i + (u_{i+1} - u_i)\frac{A'}{2^h} + (v_{i+1} - v_i)\frac{B'}{2^h} \right\} \\ &= a_{i+1} - a_i + (u_{i+1} - u_i)\min\frac{A'}{2^h} + (v_{i+1} - v_i)\max\frac{B'}{2^h} \\ &= a_{i+1} - a_i + (u_{i+1} - u_i) * 0 + (v_{i+1} - v_i) * 1 \\ &= a_{i+1} - a_i + v_{i+1} - v_i. \end{aligned}$$

Note that $B'/2^h < 1$ for any $h > 0$ and $B' < 2^h$. This is the reason why the second inequality $<$ in (12) can be replaced by $\le$.

Now suppose i is odd. Similarly, we get

$$0 \le a_{i+1} + v_{i+1},$$

$$a_{i+1} - a_i + u_{i+1} - u_i \le 0. \square$$

The condition (10) can be obtained as a consequence, using (7).

The practical improvement of the algorithm by using this exact condition is very small. We found that the average number of divisions which are simulated in one step by using (10) is 16.92, vs. 17.12 by using the exact condition, hence an improvement of only 1.2% (see Fig. 3). However, the "exact condition" approach shows which is the maximum improvement which can be achieved by trying to increase the number of divisions per step. The speed-up over the original Lehmer's algorithm is now 104% – 185%. We will try now to simplify the computations required for testing the condition.

Length	Steps per digit			Divisions per step			Time (ms)		
of operands	E	C	E/C%	E	C	E/C%	E	C	E/C%
5	0.80	0.80	100.0	17.42	17.23	101.1	0.70	0.72	97.2
10	0.90	0.91	98.9	17.24	17.04	101.2	1.70	1.72	98.8
15	0.98	0.99	99.0	17.17	16.97	101.2	2.90	2.97	97.6
20	1.00	1.01	99.0	17.11	16.91	101.2	4.23	4.30	98.4
25	1.02	1.03	99.0	17.14	16.94	101.2	5.72	5.85	97.8
30	1.03	1.04	99.0	17.13	16.94	101.1	7.37	7.53	97.9
35	1.04	1.05	99.0	17.12	16.92	101.2	9.17	9.37	97.9
40	1.05	1.06	99.1	17.13	16.93	101.2	11.13	11.35	98.1
45	1.05	1.07	98.1	17.12	16.91	101.2	13.23	13.52	97.9
50	1.06	1.07	99.1	17.13	16.92	101.2	15.48	15.83	97.8
55	1.06	1.07	99.1	17.14	16.93	101.2	17.90	18.30	97.8
60	1.06	1.07	99.1	17.12	16.93	101.1	20.47	21.00	97.5
65	1.06	1.08	98.1	17.14	16.93	101.2	23.18	23.73	97.7
70	1.07	1.08	99.1	17.11	16.91	101.2	26.05	26.67	97.7
75	1.07	1.08	99.1	17.13	16.93	101.2	29.08	29.78	97.6
80	1.07	1.08	99.1	17.14	16.93	101.2	32.25	33.05	97.6
85	1.07	1.08	99.1	17.13	16.93	101.2	35.58	36.43	97.7
90	1.07	1.08	99.1	17.15	16.94	101.2	39.05	40.03	97.6
95	1.07	1.09	98.2	17.13	16.93	101.2	42.68	43.73	97.6
100	1.07	1.09	98.2	17.11	16.91	101.2	46.50	47.63	97.6

Figure 3: Experiments with multiprecision GCD algorithm using "exact" condition (columns E) vs. Collins' condition (columns C).

3.2 An approximative condition

Note that in the context of double-digit partial cosequence computation, the condition given by Theorem 1 is not sufficient. The cofactors $u_k, v_k, u_{k+1}, v_{k+1}$ must also be smaller than a computer word (2^{32} in our case) in order to be useful. We investigate in the sequel the size of the cofactors.

For this, we use the *continuant polynomials* (see also [Knuth 81]) defined by

$$\begin{cases} P_0() = 1, \\ P_1(x_1) = x_1, \\ P_{i+2}(x_1, \ldots, x_i, x_{i+1}, x_{i+2}) = P_i(x_1, \ldots, x_k) + x_{i+2} * P_{i+1}(x_1, \ldots, x_i, x_{i+1}). \end{cases} \tag{13}$$

which are known to enjoy the symmetry

$$P_k(x_1, \ldots, x_i) = P_k(x_i, \ldots, x_1). \tag{14}$$

By comparing the recurrence relations (6) and (13) one notes

$$\begin{aligned} |u_i| &= P_{i-2}(q_2, \ldots, q_{i-1}), \\ |v_i| &= P_{i-1}(q_1, \ldots, q_{i-1}). \end{aligned} \tag{15}$$

Also, by transforming (2) into

$$a_i = a_{i+2} + q_{i+1} * a_{i+1},$$

and using $a_i > a_{i+1}$, one can prove

$$\begin{aligned} a &\geq a_i * P_i(q_i, \ldots, q_1), \\ b &\geq a_i * P_{i-1}(q_i, \ldots, q_2). \end{aligned}$$

Hence by (14) and (15) one has

$$\begin{aligned} |v_{i+1}| &\leq a/a_i, \\ |u_{i+1}| &\leq b/a_i. \end{aligned}$$

Since a_0, a_1 are double digits, we have

$$2^{2n-1} \leq a = a_0 < 2^{2n}, \quad b = a_1 < 2^{2n},$$

where n is the bit-length of the computer word (in our implementation $n = 32$). Then

$$a_{i+1} > a_{i+2} \geq 2^n,$$

$$|v_{i+1}| \leq a_0/a_i < a_0/2^n < 2^n < a_{i+1},$$

(note that $a_{i+1} \geq 2^n$ is sufficient here).

Also

$$a_i - a_{i+1} \geq a_i - q_{i+1} * a_{i+1} = a_{i+2} \geq 2^n,$$

$$|v_i| + |v_{i+1}| \leq |v_i| + q_{i+1} * |v_{i+1}| = |v_{i+2}| \leq a/a_{i+1} < a/2^n < 2^n.$$

The previous relations allow us to develop an alternative to (10), which is

$$a_{i+2} \geq 2^n. \tag{16}$$

For the practical implementation, this means that the high order digit of a_{i+2} is not zero, which is computationally inexpensive to test.

Also, note that when (16) holds, then also:

$$|v_i| \leq |v_{i+1}| \leq |v_{i+2}| \leq |v_{i+3}| \leq a/a_{i+2} < 2^n,$$

$$|u_i| \leq |u_{i+1}| \leq |u_{i+2}| \leq |u_{i+3}| \leq b/a_{i+2} < 2^n,$$

hence the sizes of the cofactors are "good" for $k = i,\ k = i+1,\ k = i+2$. Using this, we combine the "exact" and the "approximative" condition in order to obtain maximum of efficiency.

This is done as follows:

Length	Steps per digit			Divisions per step			Time (ms)		
of operands	E	C	E/C%	E	C	E/C%	E	C	E/C%
5	0.80	0.80	100.0	17.42	17.41	100.1	0.70	0.63	111.1
10	0.90	0.90	100.0	17.24	17.22	100.1	1.70	1.53	111.1
15	0.98	0.98	100.0	17.17	17.16	100.1	2.90	2.62	110.7
20	1.00	1.00	100.0	17.11	17.09	100.1	4.23	3.88	109.0
25	1.02	1.02	100.0	17.14	17.13	100.1	5.72	5.30	107.9
30	1.03	1.03	100.0	17.13	17.12	100.1	7.37	6.82	108.1
35	1.04	1.04	100.0	17.12	17.11	100.1	9.17	8.55	107.3
40	1.05	1.05	100.0	17.13	17.12	100.1	11.13	10.47	106.3
45	1.05	1.05	100.0	17.12	17.10	100.1	13.23	12.43	106.4
50	1.06	1.06	100.0	17.13	17.11	100.1	15.48	14.58	106.2
55	1.06	1.06	100.0	17.14	17.12	100.1	17.90	16.93	105.7
60	1.06	1.06	100.0	17.12	17.11	100.1	20.47	19.43	105.4
65	1.06	1.06	100.0	17.14	17.12	100.1	23.18	22.02	105.3
70	1.07	1.07	100.0	17.11	17.09	100.1	26.05	24.90	104.6
75	1.07	1.07	100.0	17.13	17.12	100.1	29.08	27.80	104.6
80	1.07	1.07	100.0	17.14	17.12	100.1	32.25	30.88	104.4
85	1.07	1.07	100.0	17.13	17.11	100.1	35.58	34.18	104.1
90	1.07	1.07	100.0	17.15	17.14	100.1	39.05	37.57	103.9
95	1.07	1.07	100.0	17.13	17.12	100.1	42.68	41.27	103.4
100	1.07	1.08	99.1	17.11	17.09	100.1	46.50	44.92	103.5

Figure 4: Benchmarks of version 2 (improved condition in DPCC): "exact" condition (columns E) vs. "combined" condition (columns C).

- In the main loop of DPCC2, the maximum i for which (16) holds is determined, and then we know $k = i$ is "good".
- After exiting the loop, we test the only second part of the condition (because $a_{i+1} \geq 2^n$ implies the first part), and if this test passes then we know $k = i + 1$ is "good".
- Full condition must be tested in order to insure $k = i + 2$ is "good".

According to our experiments, if $a_{i+2} \geq 2^n$ and $a_{i+3} < 2^n$, then the maximum "good" value of k is $k = i$ in about 10% of the cases, $k = i + 1$ in 20% of the cases, and $k = i + 2$ in 70% of the cases. The incidence of situations when $k = i + 3$ is "good" is insignificant.

In Fig.4 we present the comparative benchmarks for the algorithm using exact condition (columns E) and approximative combined with exact (columns C). One can see that the number of simulated divisions per step decreases insignificantly, but the running time improves (the speed-up is now 116% – 191%). The speed-up obtained by the two variants is shown in Fig.5. We keep the best variant as "version 2", which will be further improved in the sequel.

4 Approximative GCD computation

The final improvement is based on the following idea: instead of computing

$$A_k = u_k * A + v_k * B,$$

$$A_{k+1} = u_{k+1} * A + v_{k+1} * B,$$

which requires 4 long multiplications, compute only A_{k+1} by the above formula, and continue the algorithm with B and A_{k+1}.

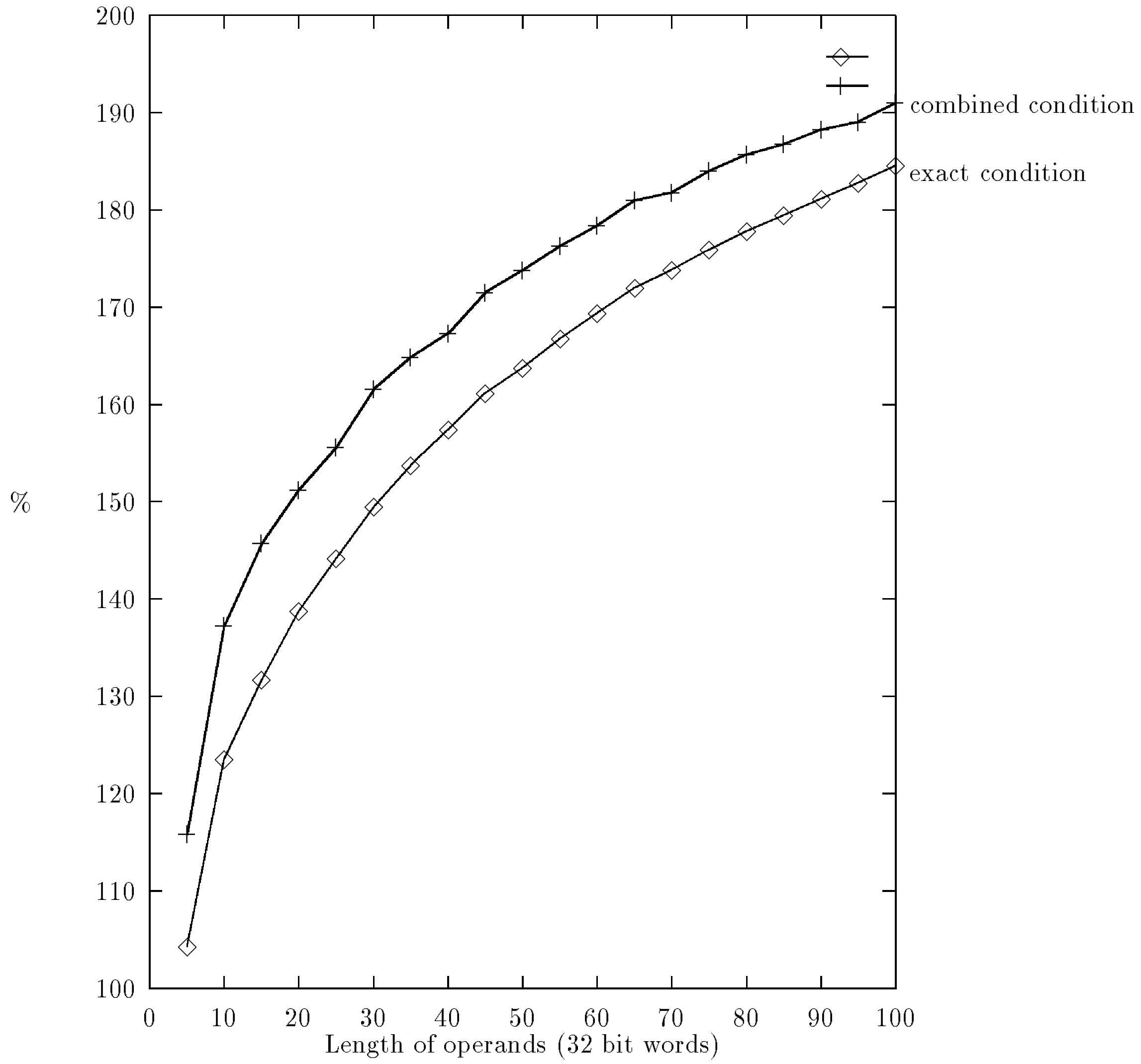

Figure 5: Speed-up for version 2.

Then, since A_{k+1} is about one word shorter than B, a division will be performed at the next step, which is computationally equivalent to one long multiplication. Hence, one step of Lehmer's scheme (4 long multiplications) will be replaced by one "half–step" and one division (3 long multiplications), reducing the number of digit multiplications by 25%.

However, this new scheme lacks in correctness, since

$$GCD(A, B) \neq GCD(B, A_{k+1}).$$

In exchange, it is easy to prove:

$$GCD(A, B) \text{ divides } GCD(B, A_{k+1}),$$

hence the final result G' of the algorithm will be, in general, bigger than the real GCD G, but still divisible by G. Then G' can be used in finding G by:

$$GCD(A, B) = GCD(A, B, G') = GCD(GCD((G', A_{\text{mod } G'}), B_{\text{mod } G'})).$$

If the "approximative" GCD G' is "near" the real GCD G, then these two GCD computations will not take long time.

One can see that the "noise" $GCD(B, A_{k+1})/GCD(A, B)$ introduced at each step is bounded by $g = GCD(u_{k+1}, B)$, because u_{k+1}, v_{k+1} are relatively prime. But it is known that for random values, the average length of the GCD is small (see, for instance, [Knuth 81]). According to our experiments, the average bit–length of the noise G'/G is 0.52 bits per step. This means 100 steps will give an average noise of less than two digits, which requires only two steps to correct.

Fig.6 presents the speed-up obtained for the "approximative" GCD computation.

The straight forward implementation of the idea does not give a satisfactory result (the speed-up decreases to 107% – 180%), because in the original program a division is performed only when the length-difference of A, B exceeds one word. The experiments show that this happens only in about 7% of the cases, hence there is no alternation between Lehmer–type and division steps.

By setting the threshold for testing the length–difference at 24 bits, the percentage of the division steps becomes 49%, hence the desired alternation is realized. The speed–up improves a little (116% – 189%), but it is still not satisfactory, probably because of the additional time consumed for calling the division routine.

The final version is obtained by replacing the calls to the division routine $A_{\text{mod } B}$ (in cases when the length–difference is at most one word), with an explicit computation of $A - q * B$, where q is the quotient of the most-significant double-digits of A and B, which has the property

$$q - 1 \leq \lfloor A/B \rfloor \leq q,$$

(see [Knuth 81]). Hence $|A - q * B|$ equals either $A_{\text{mod } B}$ or $A - A_{\text{mod } B}$, which are both good for continuing the GCD computation.

The variant obtained in this way is not much better than the previous one, (speed-up 112% – 207%) but at least gives some improvement over version 2, and also, for large lengths (80 digits), overcomes the "psychological barrier" of two times speed–up in comparison with the original Lehmer algorithm. This last variant is kept as "version 3" of the algorithm.

Fig.7 presents the speed–up for the 3 versions.

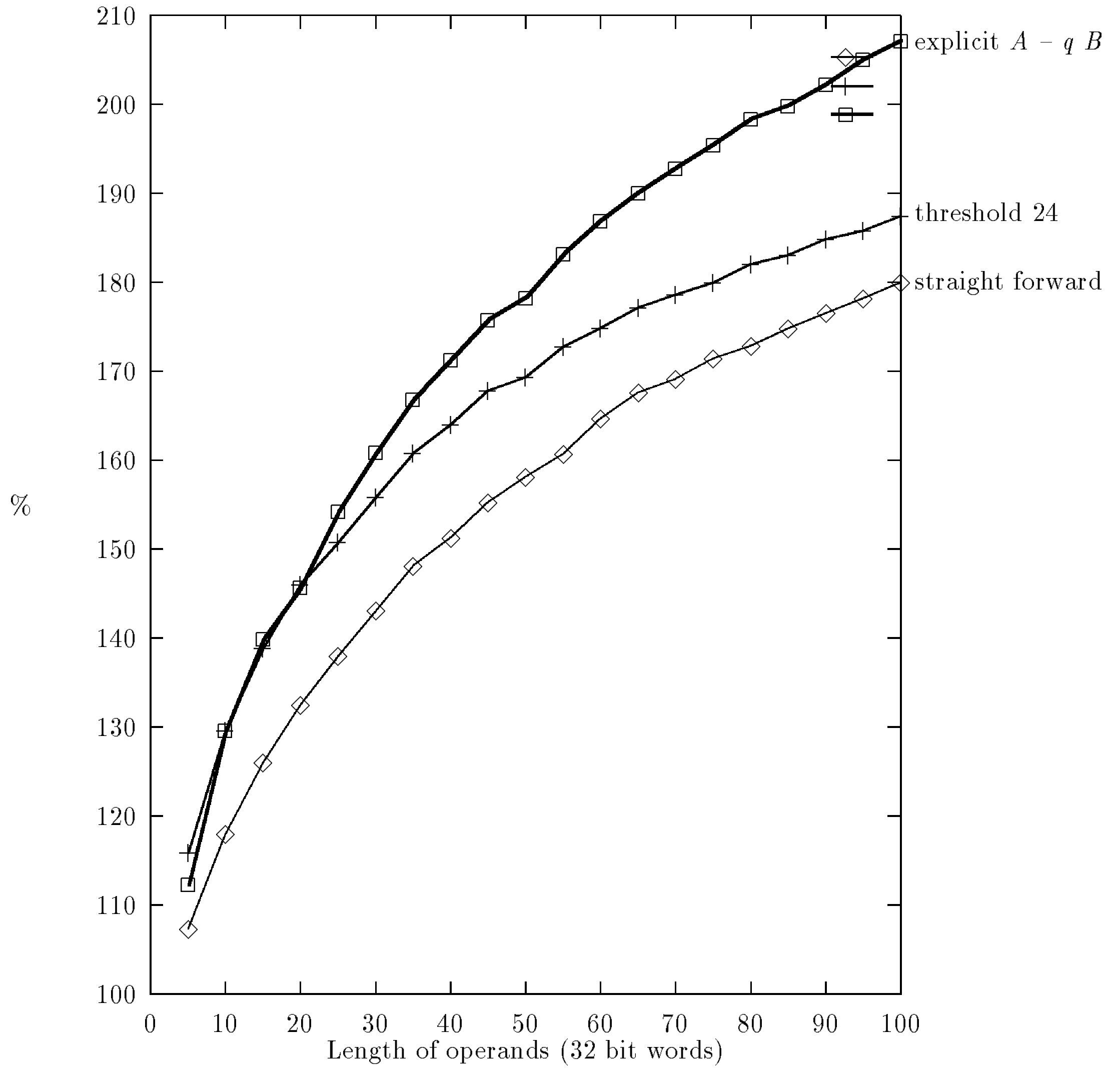

Figure 6: Speed-up for version 3.

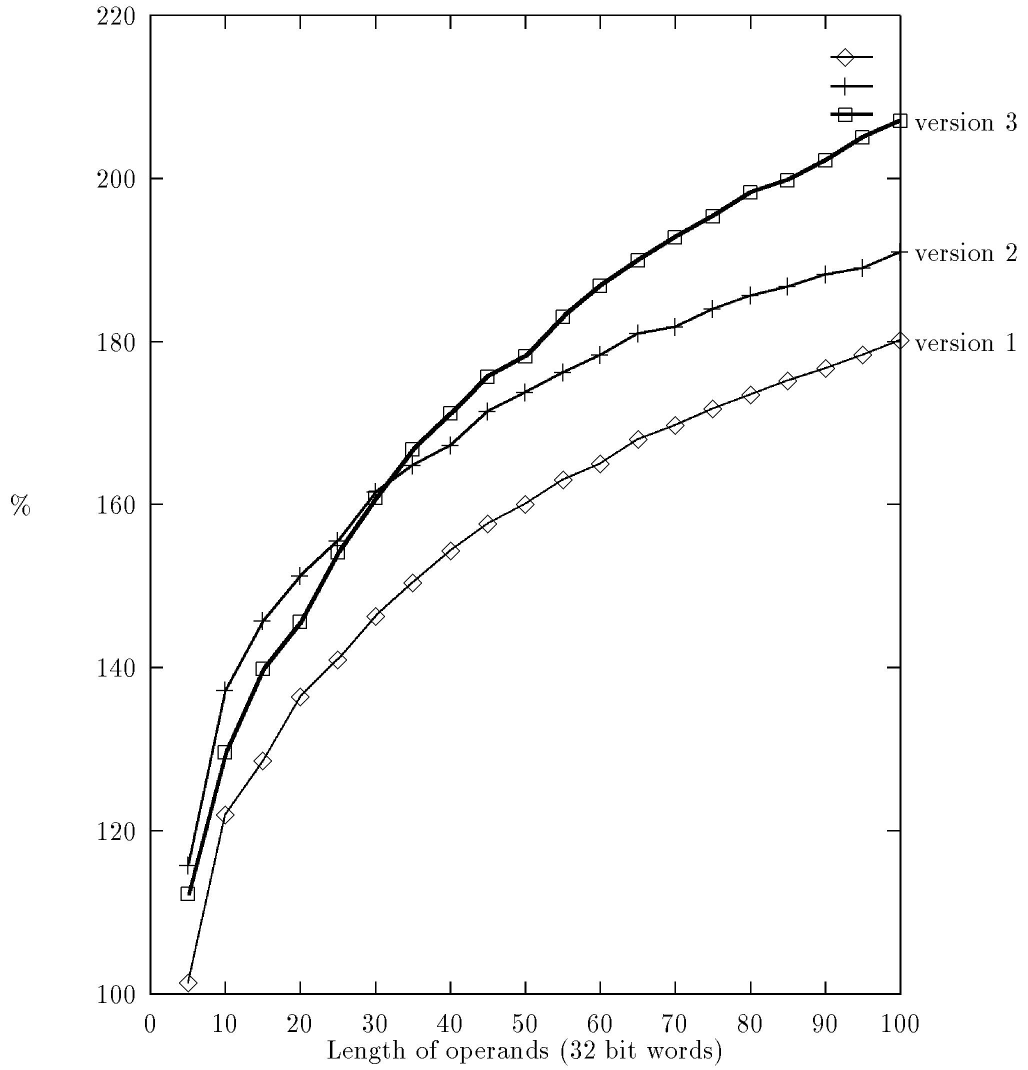

Figure 7: Speed-up for the three main versions.

References

[**Buchberger 85**] B. Buchberger — *Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory*, in N. K. Bose (ed.) — *Multidimensional Systems Theory*, D. Reidel Publishing Co., 1985.

[**Buchberger, Jebelean 92**] B. Buchberger, T. Jebelean — *Parallel Rational Arithmetic for Computer Algebra systems: Motivating Experiments*, RISC–Linz Report 92-29, May 1992.

[**Collins 80**] G. E. Collins — *Lecture notes on arithmetic algorithms*, Univ. of Wiscousin, 1980.

[**Granlund 91**] T. Granlund — *GNU MP: The GNU multiple precision arithmetic library*, Free Software Foundation, 1991.

[**Knuth 81**] D. E. Knuth — *The art of computer programming*, Vol. 2, Addison-Wesley 1981.

[**Lehmer 38**] D. H. Lehmer — *Euclid's algorithm for large numbers*, Am. Math. Mon. 45 (1938), 227–233.

[**Moenck 73**] R. T. Moenck — *Fast computation of GCDs*, Proceedings ACM V^{th} Symp. Theory of Computing, ACM 1973, 142 – 151.

[**Schönhage 71**] A. Schönhage — *Schnelle Berechung von Kettenbruchentwicklugen*, Acta Informatica 1 (1971), 139 – 144.