J. Symbolic Computation (1995) **19**, 145–157

A Double-Digit Lehmer-Euclid Algorithm for Finding the GCD of Long Integers

TUDOR JEBELEAN

Research Institute for Symbolic Computation, A-4040 Linz, Austria

(tjebelea@risc.uni-linz.ac.at)

(Received 31 January 1994)

The use of pairs of double digits in the Lehmer-Euclid multiprecision GCD algorithm halves the number of long multiplications, but a straightforward implementation of this idea does not give the desired speed-up. We show how to overcome the practical difficulties by using an enhanced condition for exiting the partial cosequence computation. Also, additional speed-up is achieved by approximative GCD computation. The combined effect of these improvements is an experimentally measured speed-up by a factor of 2 for operands with 100 32-bit words.

1. Introduction

For long integers which occur in typical applications (up to 100 computer words), it is generally accepted that the best scheme for computing the greatest common divisor (GCD) is the Lehmer-Euclid multiprecision Euclidean algorithm (Lehmer, 1938; Knuth, 1981; Collins, 1980), although recent research on generalizing the binary GCD algorithm of Stein (1967) found faster schemes (Jebelean, 1993b; Sorenson, 1994; Weber, 1993). FFT based algorithms (Schönhage, 1971; Moenck, 1973), although having a better theoretical complexity, are less efficient for integers in this range (Schönhage, 1994, page 244; Jebelean, 1993a).

We present here three ways of improving the Lehmer-Euclid algorithm: partial cosequence computation using pairs of *double digits*, a *better criterion* for ending the partial cosequence computation, and *approximative GCD computation*. The combined effect of these improvements is a speed-up by a factor of 2 over the (currently used) straightforward implementation of the Lehmer GCD algorithm.

The speed of multiprecision arithmetic has a decisive influence on the speed of computer algebra systems. By performing a series of measurements of typical algebraic algorithms (Buchberger and Jebelean, 1992), we found out that in Gröbner Bases computation (Buchberger, 1985), when increasing the input coefficient length from 1 to 10 decimal

† 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).

0747–7171/95/010145+13 $08.00/0

digits, the proportion of rational operations grows from 43% to 93%, and the total computation time grows by a factor of 12. Also, we noticed that GCD computation is the most time-consuming operation on long integers (24% to 60% of the total computing time). Neun and Melenk (1990) also note that 80% of Gröbner Bases computing time is spent in long integer arithmetic.

2. The multiprecision Euclidean algorithm

We present here the Euclidean algorithm and the multiprecision version of it (Lehmer, 1938) and we notice two useful properties of the cosequences. More details are given by Knuth (1981) and Collins (1980). We adopt here (with small modifications) the notations used by Collins (1980).

Let $A > B > 0$ be integers. The Euclidean algorithm consists of computing the *remainder sequence* of (A, B): $(A_0, A_1, \ldots, A_n, A_{n+1})$ defined by the relations

$$A_0 = A, \quad A_1 = B, \quad A_{i+2} = A_i \bmod A_{i+1}, \quad A_{n+1} = 0. \tag{2.1}$$

It is well known that $A_n = GCD(A, B)$.

The extended Euclidean algorithm consists of 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 $(V_0, V_1, \ldots, V_{n+1})$:

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

Then the following hold:

$$A_{i+2} = A_i - Q_{i+1} A_{i+1}, \tag{2.2}$$

$$A_i = AU_i + BV_i. \tag{2.3}$$

It is useful to note that the signs of the elements of each cosequence alternate:

$$\text{if } i \text{ even, then } \quad U_{i+1}, V_i \leq 0 \text{ and } \quad U_i, V_{i+1} \geq 0, \tag{2.4}$$

which leads to

$$(|U_{i+2}|, |V_{i+2}|) = (|U_i|, |V_i|) + Q_{i+1}(|U_{i+1}|, |V_{i+1}|). \tag{2.5}$$

These relations are useful for implementing the algorithm when the largest 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

$$|V_i| \geq Q_1 |U_i|, \quad \text{for all } i \geq 1. \tag{2.6}$$

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

$$|V_{i+2}| = |V_i| + Q_{i+1}|V_{i+1}| \geq Q_1|U_i| + Q_{i+1}Q_1|U_{i+1}| = Q_1|U_{i+2}|.$$

When A and B are multiprecision integers (several computer words are needed for storing the values), the divisions in (2.1) are quite time consuming. Lehmer (1938) noticed

that several steps of the Euclidean algorithm can be simulated by using only single-precision divisions. The idea is to apply the extended Euclidean 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 as big as possible, but still single 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 $(u_0, \ldots, u_{k+1})$ and $(v_0, \ldots, v_{k+1})$, for some $k \geq 0$ such that

$$q_i = Q_i, \quad \text{for all } i \leq k. \tag{2.7}$$

This process is called *digit partial cosequence calculation* (DPCC). When (2.7) holds, 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 (2.3):

$$A_k = u_k A + v_k B, \; A_{k+1} = u_{k+1} A + v_{k+1} B, \tag{2.8}$$

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

However, the condition (2.7) 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 (2.7).

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

Collins (1980) developed a better condition which needs only the computation of the sequences (q_i), (a_i) and (v_i), namely

$$a_{i+1} \geq |v_{i+1}| \quad \text{and} \quad a_{i+1} - a_i \geq |v_{i+1} - v_i|, \quad \text{for all } i \leq k. \tag{2.9}$$

This condition has the advantage that only one quotient sequence has to be computed, and only one of the cosequences. As noted by G. H. Bradley (acc. to Knuth, 1981), u_k, u_{k+1} can be found at the end of partial cosequence computation, by using

$$u_k a + v_k b = a_k, \; u_{k+1} a + v_{k+1} b = a_{k+1}. \tag{2.10}$$

As a comparison base for our improvements, we implemented this version of the multiprecision Euclidean algorithm using the GNU multiprecision library (Granlund, 1991) and the GNU optimizing compiler on a Digital DECstation 5000/200. 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 presented in this paper.

3. The double-digit algorithm

Recovering A_k, A_{k+1} by (2.8) involves 4 multiplications of single-digit numbers (the cofactors) by multidigit numbers (A and B), 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 cosequence computation (DPCC) would be performed on double digits, then each cofactor will still fit into one word. The recovering step will require the same computational effort, but will occur (roughly) two times less frequently. This is the main idea of the improvement of the Lehmer algorithm which we will discuss in this section.

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
25	2.28	1.03	221.4	7.58	16.94	44.7	8.25	13.30	62.0
50	2.38	1.07	222.4	7.57	16.92	44.7	25.35	31.32	80.9
75	2.41	1.08	223.1	7.57	16.93	44.7	51.17	53.03	96.5
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 single-digit DPCC (columns S) vs. double-digit DPCC2 (columns D).

A straightforward implementation of this idea is not very efficient (see Fig.1): the experimental speed-up ranges from 38% (5-word operands) to 109% (100-word operands), hence is rather a slowdown. However, the average values in the table show that, indeed, the number of divisions that 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-digit operations involved. Therefore we will concentrate in the sequel on improving the partial cosequence computation for double words (DPCC2). A few simple improvements lead to a speed-up of almost 2:

(A1) Some double-digit operations can be reduced to single-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 (2.5) and (2.6). 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%.

(A2) Digit partial cosequence computation for single words (DPCC) can be used at the beginning of DPCC2. The speed-up grows to 58% - 140%.

(A3) Since only the second cosequence (v_i) is needed for testing the exit condition (2.9), the cofactors u_k, u_{k+1} can be computed at the end, using (2.10). However, our experiments show no improvement by this method: the speed-up decreases to 53% – 133%.

(A4) Because the quotients are usually small (Knuth, 1981), 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 experiments 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%. Note that simulating division by subtraction does not give a speed-up in the original DPCC, because the single-precision division is fast enough.

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

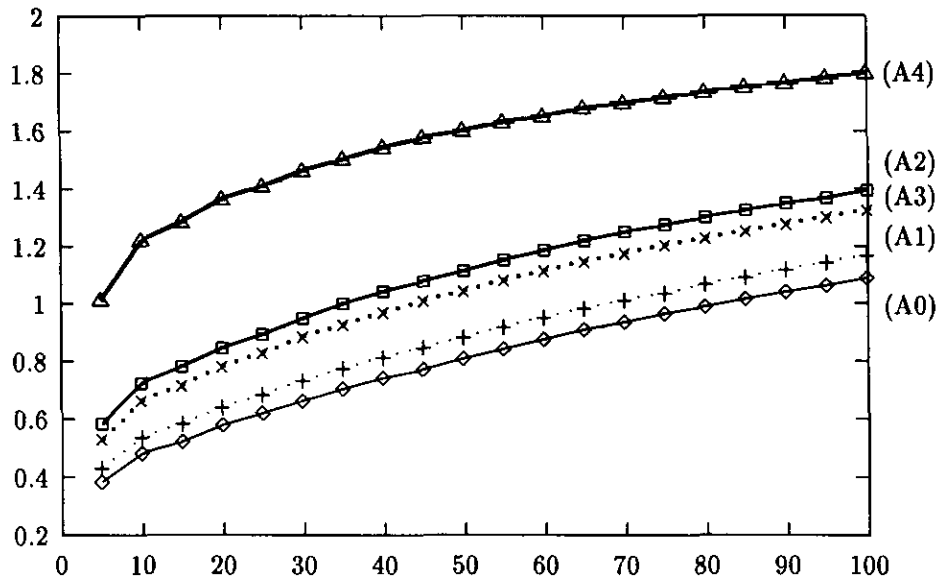

Figure 2. Speed-up for successive improvements of version A (length of inputs from 5 to 100 32-bit words).

4. On the condition needed in digit partial cosequence computation

We develop here an "exact" condition for continuing the digit partial cosequence computation, that is, a condition which precisely indicates which is the highest k for which (2.7) is satisfied, when only a and b are given. 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.

4.1. An exact condition

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

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

if and only if

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

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

$$(A, B) = (A_0, A_1) = (2^h a + A', \ 2^h b + B').$$

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

$$\begin{aligned} 0 \;\leq\; & a_{i-1} + u_{i-1}\tfrac{A'}{2^h} + v_{i-1}\tfrac{B'}{2^h} - q_i\left(a_i + u_i\tfrac{A'}{2^h} + v_i\tfrac{B'}{2^h}\right) \;< \\ & < \; a_i + u_i\tfrac{A'}{2^h} + v_i\tfrac{B'}{2^h}, \quad \forall h, A', B'. \end{aligned} \tag{4.1}$$

Using

$$u_{i+1} = u_{i-1} - q_i u_i, \; v_{i+1} = v_{i-1} - q_i v_i,$$

one notes that (4.1) is equivalent to the conjunction of

$$0 \;\leq\; \min\left\{ a_{i+1} + u_{i+1}\frac{A'}{2^h} + v_{i+1}\frac{B'}{2^h} \;\middle|\; 0 < h, \; 0 \leq A', B' < 2^h \right\} \tag{4.2}$$

and

$$\max\left\{ a_{i+1} - a_i + (u_{i+1} - u_i)\frac{A'}{2^h} + (v_{i+1} - v_i)\frac{B'}{2^h} \,\middle|\, 0 < h, \; 0 \leq A', B' < 2^h \right\} \;\leq\; 0. \tag{4.3}$$

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

Suppose i is even. Then by (2.4):

$$u_{i+1}, v_i \leq 0, \quad u_i, v_{i+1} \geq 0,$$

hence the right hand side of (4.2) becomes

$$\min\left\{ a_{i+1} + u_{i+1}\frac{A'}{2^h} + v_{i+1}\frac{B'}{2^h} \right\} \;=\; a_{i+1} + u_{i+1}\max\frac{A'}{2^h} + v_{i+1}\min\frac{B'}{2^h}$$

$$=\; a_{i+1} + u_{i+1} * 1 + v_{i+1} * 0 \;=\; a_{i+1} + u_{i+1}.$$

Also, the left hand side of (4.3) equals

$$\max\left\{ a_{i+1} - a_i + (u_{i+1} - u_i)\frac{A'}{2^h} + (v_{i+1} - v_i)\frac{B'}{2^h} \right\}$$

$$=\; a_{i+1} - a_i + (u_{i+1} - u_i)\min\frac{A'}{2^h} + (v_{i+1} - v_i)\max\frac{B'}{2^h}$$

$$=\; a_{i+1} - a_i + (u_{i+1} - u_i) * 0 + (v_{i+1} - v_i) * 1 \;=\; a_{i+1} - a_i + v_{i+1} - v_i.$$

Note that $B'/2^h$ is strictly smaller than 1 for any $h > 0$ and $B' < 2^h$. This is the reason why the second inequality $<$ in (4.1) can be replaced by $\leq$.

Now suppose i is odd. Similarly, one obtains

$$0 \leq a_{i+1} + v_{i+1}, \; a_{i+1} - a_i + u_{i+1} - u_i \leq 0. \; \square$$

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
25	1.02	1.03	99.0	17.14	16.94	101.2	5.72	5.85	97.8
50	1.06	1.07	99.1	17.13	16.92	101.2	15.48	15.83	97.8
75	1.07	1.08	99.1	17.13	16.93	101.2	29.08	29.78	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).

Remark: The condition given by the theorem is exact in the sense that it gives the maximal number of iterations when h, A', and B' are not known. As one of the referees correctly pointed out, for some particular values of these variables the number of correct iterations may be higher.

The condition (2.9) can be obtained as a consequence, using (2.6).

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 (2.9) 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 algorithm becomes 104% – 185%. We will try now to simplify the computations required for testing the condition.

4.2. 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* (Knuth, 1981, section 4.5.3) defined by

$$\begin{cases} P_0() = 1, \\ P_1(x_1) = x_1, \\ P_{i+2}(x_1, \ldots, x_{i+2}) = P_i(x_1, \ldots, x_i) + x_{i+2} * P_{i+1}(x_1, \ldots, x_{i+1}). \end{cases} \tag{4.4}$$

which are known to enjoy the symmetry

$$P_k(x_1, \ldots, x_i) = P_k(x_i, \ldots, x_1). \tag{4.5}$$

By comparing the recurrence relations (2.5) and (4.4) one notes

$$|u_i| = P_{i-2}(q_2, \ldots, q_{i-1}), \ |v_i| = P_{i-1}(q_1, \ldots, q_{i-1}). \tag{4.6}$$

Also, by transforming (2.2) into

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

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

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

Hence by (4.5) and (4.6) one has

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

Since a_0, a_1 are double digits, we have

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

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

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

Then

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

(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 (2.9), which is

$$a_{i+2} \geq 2^n. \tag{4.7}$$

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 (4.7) holds, then

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

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

hence the cofactors are at most one word long for $k = i,\ k = i+1,\ k = i+2$. It remains to determine the maximal k which is correct (in the sense that the k-length quotient sequences of a, b and A, B match). In order to do this, we combine the "exact" and the "approximative" condition as follows:

> In the main loop of DPCC2, the maximal i for which (4.7) holds is determined, and then we know $k = i$ is correct.
> After exiting the loop, we test the only right hand side of the condition given by Theorem 1 (because $a_{i+1} \geq 2^n$ implies the left hand side), and if this test passes then we know $k = i + 1$ is correct.
> Both sides of the condition must be tested in order to insure $k = i + 2$ is correct.

According to our experiments, if $a_{i+2} \geq 2^n$ and $a_{i+3} < 2^n$, then the maximum correct 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 correct is less than 1%.

In Fig.4 we present the comparative benchmarks for the algorithm using exact condition (columns B0) and the combined condition (columns B1). One can see that the number of simulated divisions per step decreases insignificantly, but the running time

Length	Steps per digit			Divisions per step			Time (ms)		
of operands	B0	B1	B0/B1%	B0	B1	B0/B1%	B0	B1	B0/B1%
5	0.80	0.80	100.0	17.42	17.41	100.1	0.70	0.63	111.1
25	1.02	1.02	100.0	17.14	17.13	100.1	5.72	5.30	107.9
50	1.06	1.06	100.0	17.13	17.11	100.1	15.48	14.58	106.2
75	1.07	1.07	100.0	17.13	17.12	100.1	29.08	27.80	104.6
100	1.07	1.08	99.1	17.11	17.09	100.1	46.50	44.92	103.5

Figure 4. Benchmarks of version B (improved condition in DPCC): "exact" condition (columns B0) vs. "combined" condition (columns B1).

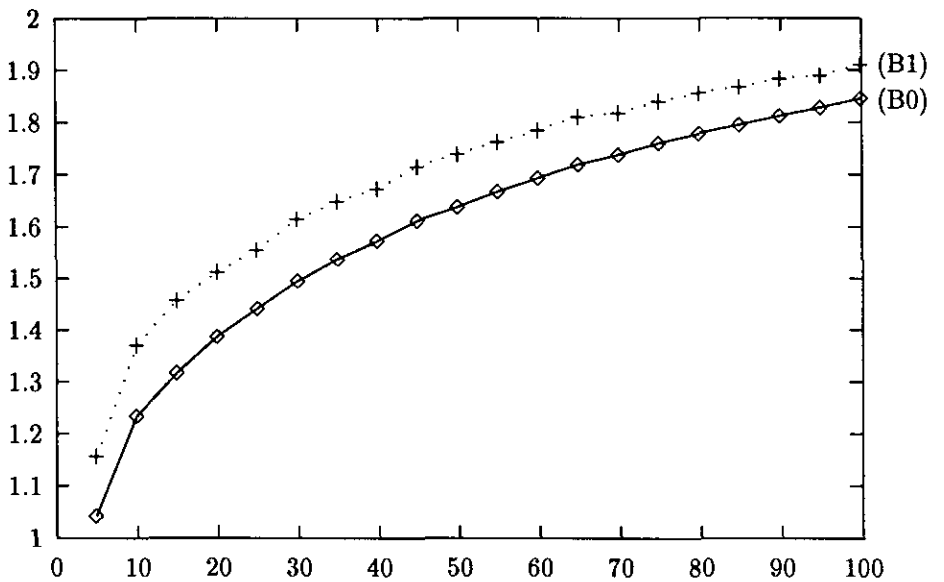

Figure 5. Speed-up for successive improvements of version B (length of inputs from 5 to 100 32-bit words).

improves: the speed-up is now 116% – 191% (see Fig.5). We keep the best variant as "version B", which will be further improved in the sequel.

5. Approximative GCD computation

The final improvement is based on the following idea: instead of computing (2.8), which requires 4 long multiplications, compute only A_{k+1} and continue the algorithm with B and A_{k+1}. 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 the Lehmer 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%.

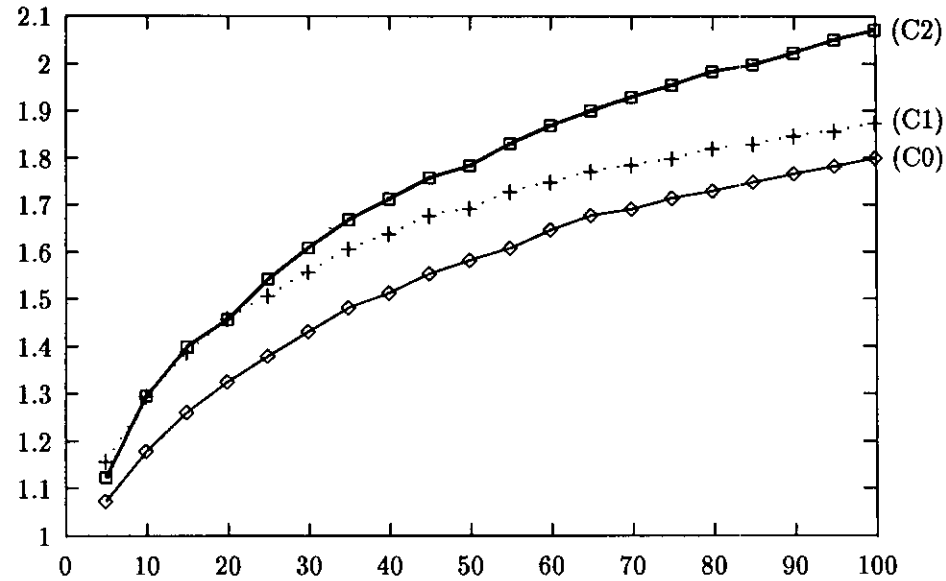

Figure 6. Speed-up for successive improvements of version C (length of inputs from 5 to 100 32-bit words).

However, this new scheme lacks correctness, since $GCD(A, B)$ may properly divide $GCD(B, A_{k+1})$. Hence the final result G' of the algorithm will be, in general, bigger than the true GCD G, but still divisible by G. Then G' can be used in finding G by:

$$GCD(A, B) = GCD(A, B, G') = GCD(A \bmod G', GCD(B \bmod G', G')).$$

If the "approximative" GCD G' is "near" the real GCD G, then these two GCD computations will be inexpensive.

Experimentally (20,000 GCD computations on random operands ranging from 5 to 100 32-bit words) we detected an average noise of 0.52 bits per step. This means 100 steps will give an average noise of less than two digits, whose correction requires only four steps of the original Lehmer-Euclid scheme. Fig.6 presents the speed-up obtained for the "approximative" GCD computation:

(C0) The straightforward 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.

(C1) 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 required for calling the division routine.

(C2) The final variant is obtained by replacing the calls to the division routine A mod B (in cases when the length-difference is at most one word), with an explicit computation of $A - qB$, where q is the quotient of the most significant double digits of A and B, which

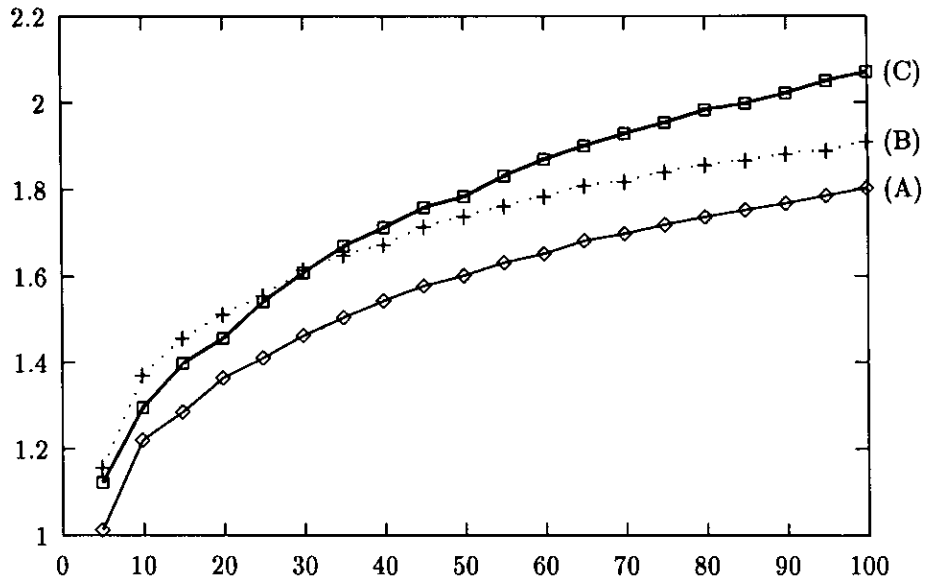

Figure 7. Speed-up for the three main versions, depending on the length of inputs (in 32-bit words).

has the property

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

Hence $|A - q * B|$ equals either $(A \bmod B)$ or $(A - A \bmod B)$, which are both good for continuing the GCD computation. This subvariant performs a little better than the previous one (speed-up 112% – 207%), it gives some improvement over version B, and also, for large lengths (80 words), overcomes the "psychological barrier" of two times speed-up in comparison with the original Lehmer algorithm. This last variant is kept as "version C" of the algorithm.

Fig.7 presents the speed-up for the 3 main versions. The absolute timings (on DECstation 5000/200) range from 0.65 ms to 41.43 ms for the improved algorithm, versus 0.73 ms to 85.82 ms for the original Lehmer-Euclid algorithm.

In fact, the speed-up increases for bigger inputs. We obtained 230% speed-up for 300 word operands on a DECstation 5000/240 (see Fig.8).

Conclusions

Starting from the idea of using double digits in digit partial cosequence computation, and applying various improvements, one can speed-up the multiprecision Lehmer-Euclid algorithm by a factor of 2.

Although some of the improvements are related to the particular hardware used for the experiments (DECstation 5000/200, DECstation 5000/240), most of the ideas will work for any sequential machine (a correction of thresholds might be necessary).

Until recently, the Lehmer-Euclid algorithm was considered the most efficient for practical applications. However, new research on improving the binary algorithm of Stein

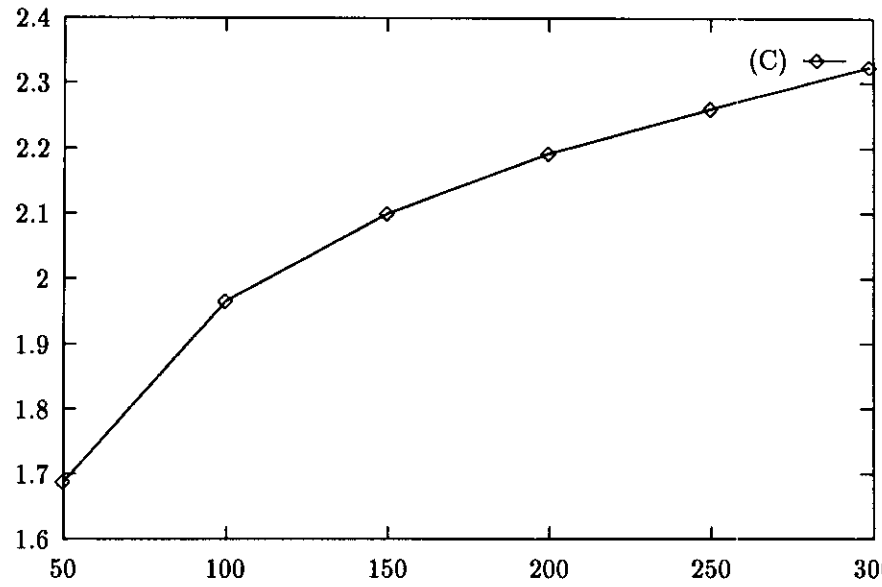

Figure 8. Speed-up for version (C) on very long inputs (50 to 300 words of 32 bits).

(1967) found faster GCD algorithms (Jebelean, 1993b; Sorenson, 1994; Weber, 1993). The improved Lehmer-Euclid method presented here is in the same range as these ones.

A natural extension of this double-digit Lehmer-Euclid scheme would be the extended Euclidean algorithm. Currently Collins and Encarnación (1994) are improving the extended Euclidean algorithm and the multiprecision integer-to-rational algorithm of Wang *et al.* (1982) by using a Lehmer-like approach and double-digit DPCC.

References

Buchberger, B., Jebelean, T. (1992). Parallel rational arithmetic for Computer Algebra Systems: Motivating experiments. In *3rd Scientific Workshop of the Austrian Center for Parallel Computation.* Report ACPC/TR 93-3, February 1993.

Buchberger, B. (1985). Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory. In Bose, Reidel, editors, *Recent trends in Multidimensional Systems*, pages 184–232. D. Reidel Publishing Company, Dordrecht-Boston-Lancaster.

Collins, G. E., Encarnación, M. J. (1994). Efficient rational number reconstruction. Technical report, RISC-Linz. To appear.

Collins, G. E. (1980). Lecture notes on arithmetic algorithms. Univ. of Wisconsin.

Granlund, T. (1991). GNU MP: The GNU multiple precision arithmetic library. Distributed by the Free Software Foundation.

Jebelean, T. (1993). A generalization of the binary GCD algorithm. In Bronstein, M., editor, *ISSAC'93: International Symposium on Symbolic and Algebraic Computation*, pages 111–116. ACM Press.

Jebelean, T. (1993). Comparing several GCD algorithms. In Swartzlander, E., Irwin, M. J. and Jullien, G., editors, *ARITH-11: IEEE Symposium on Computer Arithmetic*, pages 180–185. IEEE Computer Society Press.

Knuth, D. E. (1981). *The art of computer programming*, volume 2. Addison-Wesley, 2 edition.

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

Moenck, R. T. (1973). Fast computation of GCDs. In *ACM V^{th} Symp. Theory of Computing*, pages 142–151. ACM press.

Neun, W., Melenk, H. (1990). Very large Gröbner basis calculations. In Zippel, editor, *Computer algebra and parallelism. Proceedings of the second International Workshop on Parallel Algebraic Computation*, pages 89–100. LNCS 584, Springer Verlag.

Schönhage, A. (1971). Schnelle Berechung von Kettenbruchentwicklugen. *Acta Informatica*, 1:139–144.

Schönhage, A., Grotefeld, F. W., Vetter, E. (1971). *Fast Algorithms. A Multitape Turing Machine Implementation*. B.I. Wissenschaftsverlag, Manheim.

Sorenson, J. (1994). Two fast GCD algorithms. *J. of Algorithms*, 16:110–144.

Stein, J. (1967). Computational problems associated with Racah algebra. *J. Comp. Phys.*, 1:397–405.

Wang, P. S., Guy, M. J. T., Davenport, J. H. (1982). P-adic reconstruction of rational numbers. *ACM SIGSAM Bulletin*, 16(2):2–3.

Weber, Ken (1993). The accelerated integer GCD algorithm. Tech. Rep. ICM-9307-55, Kent State University. To appear in ACM Trans. on Math. Software.