A Generalization of the Binary GCD 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

A generalization of the binary algorithm for operation at "word level" by using a new concept of "modular conjugates" computes the GCD of multiprecision integers two times faster than Lehmer–Euclid method. Most importantly, however, the new algorithm is suitable for systolic parallelization, in "least-significant digits first" pipelined manner and for aggregation with other systolic algorithms for the arithmetic of multiprecision rational numbers.

Introduction

Computation of the Greatest Common Divisor (GCD) of long integers is heavily used in Computer Algebra Systems, because it occurs in normalization of rational numbers and other important subalgorithms. According to the experiments in [Buchberger, Jebelean 92], in typical algebraic computations more than half of the time is spent for calculating GCD of long integers. For instance, in Gröbner Bases computation [Buchberger 85], calculating GCD takes 53% of the total time if the length of the input coefficients is 5 decimal digits and 70% if the length is 50.

For the range of integers which typically occur in algebraic computations (up to 100 words of 32 bits), the asymptotically fast GCD algorithms based on FFT multiplication scheme [Schönhage 71], [Moenck 73] do not have an efficient implementation. Therefore, it seems to be generally accepted that the Lehmer–Euclid algorithm [Lehmer 38], [Collins 80], [Knuth 81] is the best one for practical applications, although the binary GCD algorithm [Stein 67], [Brent 76], [Knuth 81] can be efficiently adapted to multidigit computation in the same way Lehmer improved the Euclidean algorithm [Knuth 81], and then it gives about 1.45 speed–up over Lehmer–Euclid GCD [Jebelean 92c].

*Acknowledgements:
Austrian Forschungsförderungsfonds, project S5302-PHY (Parallel Symbolic Computation);
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);

The **binary** GCD algorithm consists in making the two operands odd by binary–shifting, and then, since their *least–significant bits* are equal, one obtains by subtraction a number whose *least–significant bit* is null, which by another binary–shift becomes at least one-bit shorter than the original numbers. One iterates this process until the GCD is obtained. If by subtraction a negative number is obtained, then the next step will be in fact an addition, hence, in general, the result will not be shortened anymore. Therefore, at each step the two operands have to be compared in order to identify the right minuend. This makes the binary algorithm less suitable for adaptation to multiprecision computations, because such an adaptation means simulating several steps by using only the information contained in the least–significant words of the operands, and recovering the full–length operands only when simulation is not accurate anymore. Also, this makes the binary algorithm less suitable for parallelization (in particular, for systolic parallelization), because such parallelization is efficient if one "master processor" needs only the information contained in the least–significant words for making the computational decisions at each step. These decisions are then sent to other parallel "slave processor" which operate on the other digits of the operands.

In the **plus–minus** GCD algorithm, [Brent, Kung 83] remove this disadvantage of the binary algorithm, by noticing that, if the *least–significant double–bits* of the two (odd) operands are equal, then one can perform subtraction, otherwise addition, in order to make the *least–significant double–bit* of the result null. Hence, the shifted result is shortened no matter which are the signs of the operands. This algorithm is easier to adapt to multidigit computation (acc. to [Jebelean 92c], the performance obtained is similar to that of the Lehmer–Euclid algorithm), and it is also suitable for systolic parallelization (see [Brent, Kung 83]). However, the "binary" level at which this algorithm operates makes it suitable for hardware implementation, rather than for implementation on multiprocessor machines.

This paper presents a **generalization** of the above idea to arbitrary bit–length: starting from the *least–significant double–words* of the (odd) operands, one can always find two cofactors ("modular conjugates") which are at most one word long, such that the linear combination of the operands by these cofactors is a number whose *least–significant double–word* is null. By binary shifting one obtains a number which is one word (for instance 32 bits) shorter then the initial operands. This algorithm is more efficient for multidigit GCD computation than the straight forward adaptation of the plus–minus algorithm (experimentally 2.35 times speed–up), and, in fact, it seems to be faster than any other multiprecision algorithm, according to the experiments in [Jebelean 92c].

Most importantly, however, this algorithm is suitable for systolic parallelization in "least-significant digits first" (LSF) pipelined manner, this time at "word" level, which makes it suitable for implementation on multiprocessor machines. This makes it also suitable for pipelined aggregation with other LSF systolic multiprecision algorithms (like multiplication [Atrubin 65], exact division [Jebelean 92b]) and with pipelined units for addition/subtraction. This is very useful in building systolic algorithms for multiprecision rational arithmetic, which is extremely time consuming in typical algebraic computations. (According to the experiments in [Buchberger, Jebelean 92], in Gröbner Bases computation, 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.)

1 Modular Conjugates

Let us remember the basic Euclidean method. Starting with two positive integers $a_0 \geq a_1$, one computes the remainder sequence $(a_i)_{1 \leq i \leq n+1}$ defined by the relations:

$$a_{i+2} = a_i \bmod a_{i+1}, \quad a_{n+1} = 0. \tag{1}$$

and then one has:

$$GCD(a_0, a_1) = a_n.$$

A detailed presentation of the Euclidean algorithm can be found in [Knuth 81].

The *extended* Euclidean algorithm (also in [Knuth 81]) consists in computing additionally the *quotient sequence* $(q_i)_{1 \leq i \leq n}$ and the *cosequences* of *cofactors* $(u_i, v_i)_{0 \leq i \leq n+1}$ defined by:

$$q_{i+1} = \lfloor a_i / a_{i+1} \rfloor, \tag{2}$$

$$u_0 = 1, v_0 = 0, u_1 = 0, v_1 = 1, \tag{3}$$

$$(u_{i+2}, v_{i+2}) = (u_i, v_i) - q_{i+1} * (u_{i+1}, v_{i+1}). \tag{4}$$

Then one has:

$$a_{i+2} = a_i - q_{i+1} * a_{i+1} \tag{5}$$

$$u_i * a_0 + v_i * a_1 = a_i. \tag{6}$$

It is useful to note that the signs of the cofactors alternate. Indeed, by (3):

$$u_0 \geq 0, \quad v_0 \leq 0,$$

$$u_1 \leq 0, \quad v_1 \geq 0.$$

If one assumes that for any $i < k$:

$$u_i \geq 0, \quad v_i \leq 0, \quad \text{if } i \text{ even},$$

$$u_i \leq 0, \quad v_i \geq 0, \quad \text{if } i \text{ odd},$$

then using (4) one can deduce the above relations for $i = k \geq 2$. Hence, one has:

$$|u_i| = (-1)^i * u_i, \quad |v_i| = (-1)^{i+1} * v_i$$

and (4) can be also written as:

$$(|u_{i+2}|, |v_{i+2}|) = (|u_i|, |v_i|) + q_{i+1} * (|u_{i+1}|, |v_{i+1}|). \tag{7}$$

In order to investigate the size of the cofactors, we use the *continuant polynomials* (see also [Knuth 81]) defined by

$$\begin{cases} P_0() = 1, \\ P_1(x_1) = x_1, \\ P_{i+2}(x_1, \ldots, x_i, x_{i+1}, x_{i+2}) = P_i(x_1, \ldots, x_i) + x_{i+2} * P_{i+1}(x_1, \ldots, x_i, x_{i+1}). \end{cases} \tag{8}$$

which are known to enjoy the symmetry

$$P_i(x_1, \ldots, x_i) = P_i(x_i, \ldots, x_1). \tag{9}$$

By comparing the recurrence relations (7) and (8) one notes

$$\begin{aligned} |u_i| &= P_{i-2}(q_2, \ldots, q_{i-1}), \\ |v_i| &= P_{i-1}(q_1, \ldots, q_{i-1}). \end{aligned} \tag{10}$$

Also, by transforming (5) into

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

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

$$\begin{aligned} a_0 &\geq a_i * P_i(q_i, \ldots, q_1), \\ a_1 &\geq a_i * P_{i-1}(q_i, \ldots, q_2). \end{aligned} \tag{11}$$

Hence by (10) and (11) one has

$$\begin{aligned} |v_{i+1}| &\leq a_0/a_i, \\ |u_{i+1}| &\leq a_1/a_i. \end{aligned} \tag{12}$$

Let us denote by m the bit–length of the computer word (in our implementation $m = 32$), and let us consider two odd double–words a, b ($a, b < 2^{2m}$). We show in the sequel how to find the *modular conjugates* of a, b, that is, the integers x, y with the properties:

$$0 < x, |y| < 2^m,$$

$$(x * a + y * b) \bmod 2^{2m} = 0 \tag{13}$$

Since a, b are odd, one can find $b^{-1} \bmod 2^{2m}$ and

$$c = (a * b^{-1}) \bmod 2^{2m},$$

which is also odd. Then (13) becomes

$$(x * c + y) \bmod 2^{2m} = 0.$$

If $c < 2^m$, then $x = 1$ and $y = -c$ satisfy (13) and they also have the desired lengths. If $c \geq 2^m$, then let us apply the extended Euclidean algorithm to $a_0 = 2^{2m}$ and $a_1 = c$. Since c is odd, the remainder sequence (a_i) is strictly decreasing to $1 = GCD(2^{2m}, c)$, hence for some k:

$$0 < a_k < 2^m \leq a_{k-1}.$$

By (12):

$$|v_k| \leq a_0/a_{k-1} \leq 2^{2m}/2^m = 2^m.$$

Also, by (6):

$$2^{2m} * u_k + c * v_k = a_k,$$

hence:

$$(c * v_k - a_k) \bmod 2^{2m} = 0.$$

In fact, $|v_k|$ cannot equal 2^m, because in this case the above relation implies:

$$a_k \bmod 2^m = 0,$$

which is impossible for

$$0 < a_k < 2^m.$$

Hence, by taking $x = |v_k|$ and $y = (-1)^k a_k$, one has the desired modular conjugates.

Note that if $m = 1$, then

$$(x, y) = \begin{cases} (1, -1), & \text{if } \lfloor a/2 \rfloor = \lfloor b/2 \rfloor, \\ (1, 1), & \text{otherwise,} \end{cases}$$

which is the basic idea of the plus–minus GCD algorithm of [Brent, Kung 83].

2 The new algorithm

Now let A, B be two multidigit integers. As in the binary and plus–minus algorithms, one can skip the trailing binary zeroes in order to make A and B odd (the number of common zeroes is stored in order to be incorporated into GCD at the end of the algorithm).

Then the modular conjugates x, y of the least significant double words of A and B can be found, and the linear combination

$$C = |x * A + y * B|/2^{2m}$$

which is (roughly) one word shorter than $\max(A, B)$. Then one can make C odd by binary shifting and reiterate the procedure with C and $\min(A, B)$.

Note that if $m = 1$, then one obtains the plus–minus GCD algorithm introduced in [Brent, Kung 83].

However, when operating at word level, some further improvements are necessary. The inter–reduction by modular conjugates is efficient when (length(A) – length(B)) is small (for $m = 32$, we experimentally observed that the best threshold is 8). Otherwise, it is more efficient to bring the lengths closer by another scheme – for instance, by division. However, division is not suitable for parallelization, and it is also a relatively slow operation. It is better to apply instead the "exact division" scheme described in [Jebelean 92a], which works like this:

Let be $d = \text{length}(A) - \text{length}(B)$ and a, b the trailing d bits of A, B.
Set $c = (a * b^{-1}) \bmod 2^d$.
Then $C = (A - c * B)/2^d$ is (at least) d bits shorter than A.

Hence, the generalized binary algorithm consists in alternating the "exact division" step with "inter-reduction by modular conjugates" step. After each alternation, the two operands become (at least) one word shorter (typically one word is 32 bits). Note that two such steps need 3 multiplications of a simple precision integer by a multiprecision integer, compared with 4 such multiplications in Lehmer–Euclid method, but the reduction obtained is one word, while in Lehmer–Euclid only half–word reduction is achieved. This is the reason why the experimental running–time decreases to half.

The algorithm terminates when a 0 is obtained. If 0 is obtained after an exact division step, then $G' = B$, and if 0 is obtained after an inter-reduction step, then $G' = (A * GCD(x, y))/y = (B * GCD(x, y))/x$, where G' is the *approximative* GCD of initial A, B.

However, G' is in general different from $G = GCD(A, B)$, because

$$GCD(A, B) \mid GCD(B, x * A + y * B),$$

but not the other way around. A "noise" factor may be introduced at each inter-reduction step, which equals

$$GCD(B/GCD(A, B), x).$$

The combined noise must be eliminated after finding the final G' by:

$$GCD(A, B) = GCD(G', A, B) = GCD(GCD(G', A \bmod G'), B \bmod G')). \qquad (14)$$

This "noise" is nevertheless small in the average case, and we experimentally observed that the operations (14) take less than 5% of the GCD computation, in the average.

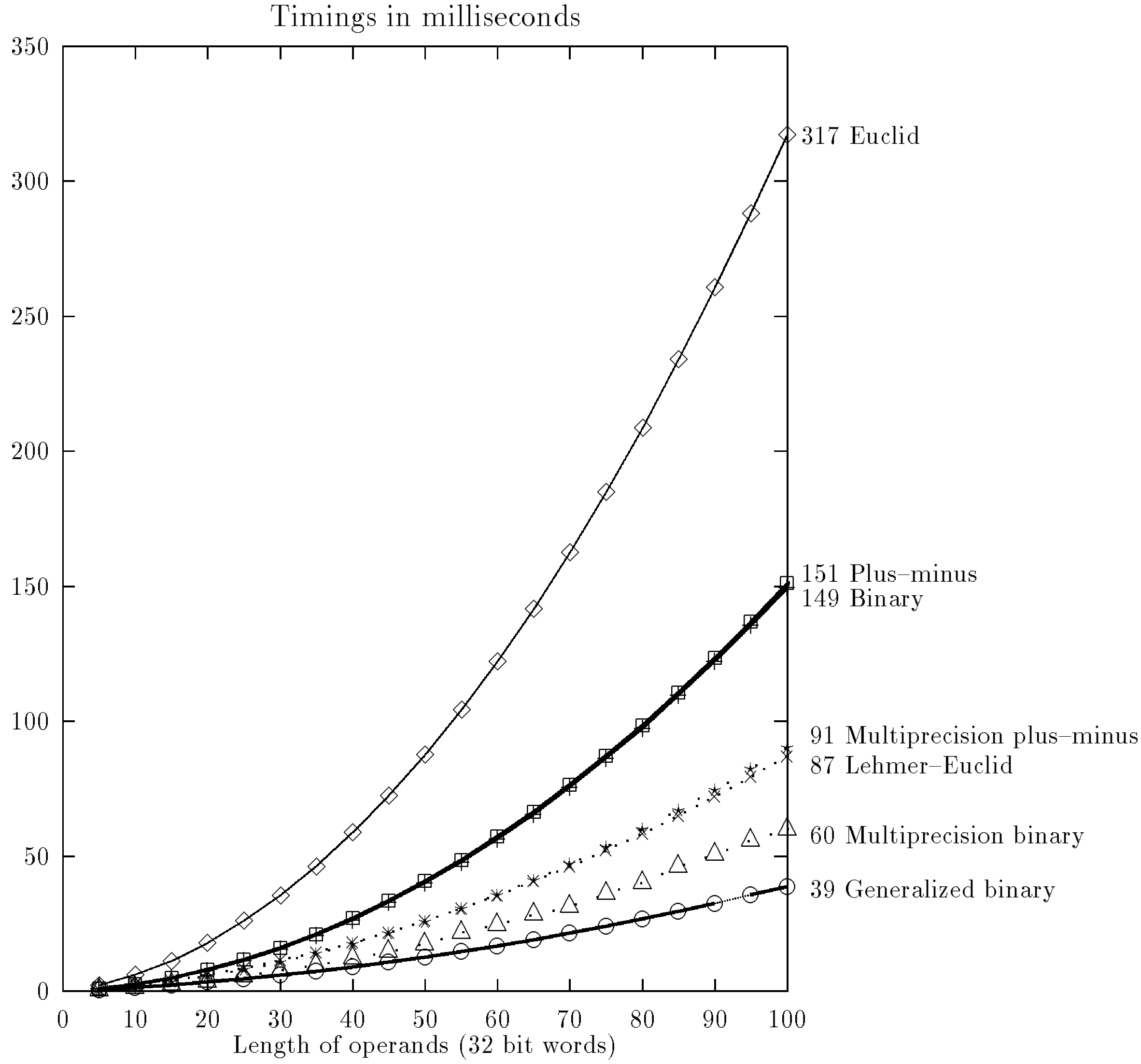
Let us also note that the computation of $x^{-1} \bmod 2^{2m}$, which is quite costly when performed via the extended Euclidean algorithm, was implemented using a scheme developed in [Jebelean 92a]:

Let be $x = x_1 * \beta + x_0$. Then:

$$x^{-1} \bmod \beta^2 = (((1 - x * a') * x') + x') \bmod \beta^2,$$

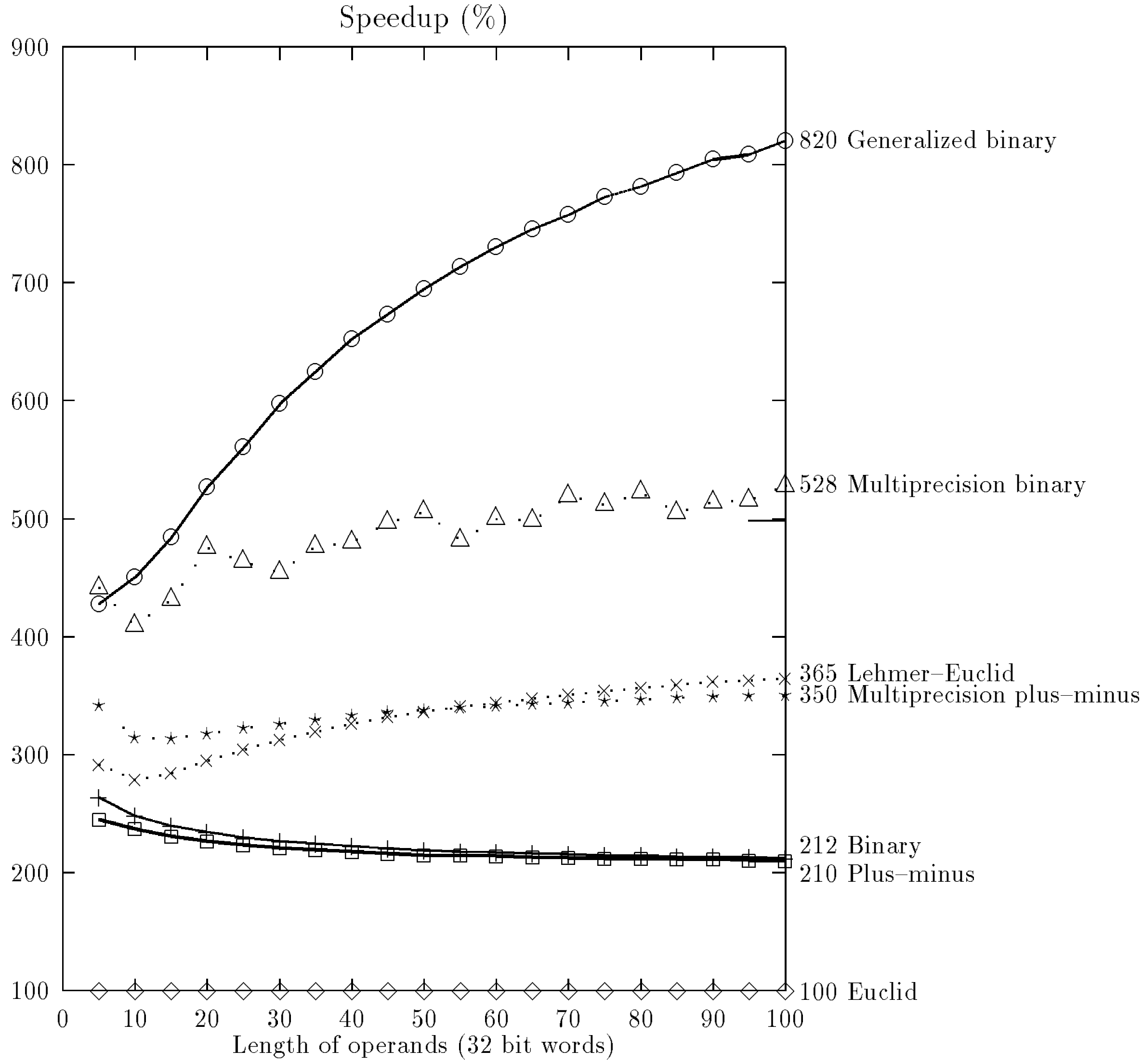
$$\text{where} \quad x' = x_0^{-1} \bmod \beta$$

Using this relation, the computation of modular inverse of a double–word can be reduced to the modular inverse of a half–word, which is done by look–up in a precomputed table.



3 Practical experiments

The new algorithm was implemented using the GNU multiprecision arithmetic library [Granlund 91] and the GNU optimizing C compiler on a Digital DECstation 5200. For comparison purposes, the Euclidean and Lehmer–Euclid, as well as binary, plus–minus and the multiprecision versions of these were also implemented and bench marked (for more details concerning the experiments, see [Jebelean 92c]). The diagrams present the average timings (in milliseconds) and the speed–up (in %) over the Euclidean algorithm.



4 Systolic computation

The generalized binary algorithm is suitable for systolic parallelization in least–significant digits first (LSF) pipelined manner, (the particular case $m = 2$ was implemented systolically by [Brent, Kung 83]). This is so because all the decisions on the procedure are taken using only the lowest digits of the operands. Hence, a "master" processor computes the modular conjugates by using only the information contained in the least–significant double–words of the operands, and then sends the modular conjugates to the "slave" processors, which apply the linear combination to the rest of the digits of the operands. The cofactors and the carries can be pipelined along the string of slave processors, while the master processor can start the next cycle of the algorithm as soon as the least–significant digits of the new operand are ready.

The LSF manner in which this algorithm operates makes it suitable for pipelined aggregation with other LSF systolic algorithms for multiprecision arithmetic (e.g. multiplication [Atrubin 65], exact division [Jebelean 92b]) and with pipelined units for addition/subtraction. Hence, one has all the components required for the realization of a LSF pipelined systolic device for the arithmetic of multiprecision rationals.

References

[Atrubin 65] A. J. Atrubin — *A one–dimensional iterative multiplier*, IEEE C-14 (1965) 394–399.

[Brent, Kung 83] R. P. Brent, H. T. Kung — *Systolic VLSI arrays for linear-time GCD computation*, in V. Anceau, E. J. Aas – *VLSI'83*, Elsevier (North-Holland), 1983, 145 – 154.

[Brent 76] R. P. Brent — *Analysis of the binary Euclidean algorithm*, in J. F. Traub – *New directions and recent results in algorithms and complexity*, Academic Press, 1976, 321 – 355.

[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.

[Jebelean 92a] T. Jebelean — *An algorithm for exact division*, RISC–Linz Report 92-35, May 1992, submitted to Journal of Symbolic Computation.

[Jebelean 92b] T. Jebelean — *Systolic algorithms for exact division*, submitted to PARS Workshop, Dresden, April 1993.

[Jebelean 92c] T. Jebelean — *Comparing several GCD algorithms*, submitted to 11^{th} ACM Symp. on Computer Arithmetic, Windsor, 1993.

[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.

[Stein 67] J. Stein — *Computational problems associated with Racah algebra*, J. Comp. Phys. 1 (1967), 397–405.