An $O(M(n) \log n)$ algorithm for the Jacobi symbol

Richard P. Brent[1] and Paul Zimmermann[2]

[1] Australian National University, Canberra, Australia
[2] INRIA Nancy - Grand Est, Villers-lès-Nancy, France

20 January 2010

Abstract. The best known algorithm to compute the Jacobi symbol of two n-bit integers runs in time $O(M(n) \log n)$, using Schönhage's fast continued fraction algorithm combined with an identity due to Gauss. We give a different $O(M(n) \log n)$ algorithm based on the binary recursive gcd algorithm of Stehlé and Zimmermann. Our implementation — which to our knowledge is the first to run in time $O(M(n) \log n)$ — is faster than GMP's quadratic implementation for inputs larger than about 10000 decimal digits.

1 Introduction

We want to compute the Jacobi symbol[3] $(b|a)$ for n-bit integers a and b, where a is odd positive. We give three algorithms based on the 2-adic gcd from Stehlé and Zimmermann [13]. First we give an algorithm whose worst-case time bound is $O(M(n)n^2) = \widetilde{O}(n^3)$; we call this the *cubic* algorithm although this is pessimistic since the algorithm is quadratic on average as shown in [5], and probably also in the worst case. We then show how to reduce the worst-case to $O(M(n)n) = \widetilde{O}(n^2)$ by combining sequences of "ugly" iterations (defined in Section 1.1) into one "harmless" iteration. Finally, we obtain an algorithm with worst-case time $O(M(n) \log n)$. This is, up to a constant factor, the same as the time bound for the best known algorithm, apparently never published in full, but sketched in Bach [1] and in more detail in Bach and Shallit [2] (with credit to Bachmann [3]).

The latter algorithm makes use of the Knuth-Schönhage fast continued fraction algorithm [9] and an identity of Gauss [6]. Although

[3] Notation: we write the Jacobi symbol as $(b|a)$, since this is easier to typeset and less ambiguous than the more usual $\left(\frac{b}{a}\right)$. $M(n)$ is the time to multiply n-bit numbers. $\widetilde{O}(f(n))$ means $O(f(n)(\log f(n))^c)$ for some constant $c \geq 0$.

this algorithm has been attributed to Schönhage, Schönhage himself gives a different $O(M(n)\log n)$ algorithm [10, 15] which does not depend on the identity of Gauss. The algorithm is mentioned in Schönhage's book [11, §7.2.3], but no details are given there.

With our algorithm it is not necessary to compute the full continued fraction or to use the identity of Gauss for the Jacobi symbol. Thus, it provides an alternative that may be easier to implement.

It may be possible to modify some of the other fast GCD algorithms considered by Möller [8] to compute the Jacobi symbol, but we do not consider such possibilities here. At best they would give a constant factor speedup over our algorithm.

We recall the main identities satisfied by the Jacobi symbol: $(bc|a) = (b|a)(c|a)$; $(2|a) = (-1)^{(a^2-1)/8}$; $(b|a) = (-1)^{(a-1)(b-1)/4}(a|b)$ for a, b odd; and $(b|a) = 0$ if $(a, b) \neq 1$.

Note that all our algorithms compute $(b|a)$ with b even positive and a odd positive. For the more general case where b is any integer, we can reduce to b even and positive using $(b|a) = (-1)^{(a-1)/2}(-b|a)$ if b is negative, and $(b|a) = (b+a|a)$ if b is odd.

We first describe a cubic algorithm to compute the Jacobi symbol. The quadratic algorithm in Section 2 is based on this cubic algorithm, and the subquadratic algorithm in Section 3 uses the same ideas as the quadratic algorithm but with an asymptotically fast recursive implementation.

For $a \in \mathbb{Z}$, the notation $\nu(a)$ denotes the 2-adic valuation $\nu_2(a)$ of a, that is the maximum k such that $2^k|a$, or $+\infty$ if $a = 0$.

1.1 Binary Division with Positive Quotient

Throughout the paper we use the binary division with positive quotient defined by Algorithm 1.1. Compared to the "centered division" of [13], it returns a quotient in $[1, 2^{j+1}-1]$ instead of in $[1-2^j, 2^j-1]$. Note that the quotient q is always odd.

Algorithm 1.1 BinaryDividePos

Input: $a, b \in \mathbb{N}$ with $\nu(a) = 0 < \nu(b) = j$
Output: q and $r = a + qb/2^j$ such that $0 < q < 2^{j+1}$, $\nu(b) < \nu(r)$
1: $q \leftarrow -a/(b/2^j) \bmod 2^{j+1}$ ▷ q is odd and positive
2: **return** $q, r = a + qb/2^j$.

With this binary division, we define Algorithm CubicBinary-Jacobi, where the fact that the quotient q is positive ensures that all a, b terms computed remain positive, and a remains odd, thus $(b|a)$ remains well-defined.[4]

Algorithm 1.2 CubicBinaryJacobi

Input: $a, b \in \mathbb{N}$ with $\nu(a) = 0 < \nu(b)$
Output: Jacobi symbol $(b|a)$

1: $s \leftarrow 0, \quad j \leftarrow \nu(b)$
2: **while** $2^j a \neq b$ **do**
3: $\quad b' \leftarrow b/2^j$
4: $\quad (q, r) \leftarrow \text{BinaryDividePos}(a, b)$
5: $\quad s \leftarrow (s + j(a^2 - 1)/8 + (a - 1)(b' - 1)/4 + j(b'^2 - 1)/8) \bmod 2$
6: $\quad (a, b) \leftarrow (b', r/2^j), \quad j \leftarrow \nu(b)$
7: **if** $a = 1$ **then** return $(-1)^s$ **else** return 0

Theorem 1. *Algorithm CubicBinaryJacobi is correct (assuming it terminates).*

Proof. We prove that the following invariant holds during the algorithm, if a_0, b_0 are the initial values of a, b:

$$(b_0|a_0) = (-1)^s (b|a).$$

This is true before we enter the while-loop, since $s = 0$, $a = a_0$, and $b = b_0$. For each step in the while loop, we divide b by 2^j, swap a and $b' = b/2^j$, replace a by $r = a + qb'$, and divide r by 2^j. The Jacobi symbol is modified by a factor $(-1)^{j(a^2-1)/8}$ for the division of b by 2^j, by a factor $(-1)^{(a-1)(b'-1)/4}$ for the interchange of a and b', and by a factor $(-1)^{j(b'^2-1)/8}$ for the division of r by 2^j. At the end of the loop, we have $\gcd(a_0, b_0) = a$; if $a = 1$, since $(b|1) = 1$, we have $(b_0|a_0) = (-1)^s$, otherwise $(b_0|a_0) = 0$. □

Lemma 1. *The quantity $a+2b$ is non-increasing in Algorithm CubicBinaryJacobi.*

[4] Möller says in [8]: "*if one tries to use positive quotients* $0 < q < 2^{k+1}$*, the* [binary gcd] *algorithm no longer terminates*". However, with a modified stopping criterion as in Algorithm CubicBinaryJacobi, the algorithm terminates (we prove this below).

Proof. At each iteration of the "while" loop, a becomes $b/2^j$, and b becomes $(a + qb/2^j)/2^j$. In matrix notation

$$\begin{pmatrix} a \\ b \end{pmatrix} \leftarrow \begin{pmatrix} 0 & 1/2^j \\ 1/2^j & q/2^{2j} \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix}. \tag{1}$$

Therefore $a + 2b$ becomes

$$\frac{b}{2^j} + 2\left(\frac{a + qb/2^j}{2^j}\right) = \frac{2a}{2^j} + (1 + 2q/2^j)\frac{b}{2^j}. \tag{2}$$

Since $j \geq 1$, the first term is bounded by a. In the second term, $q \leq 2^{j+1} - 1$, thus the second term is bounded by $(5/2^j - 2/2^{2j})b$, which is bounded by $9b/8$ for $j \geq 2$, and equals $2b$ for $j = 1$. □

If $j \geq 2$, then $a + 2b$ is multiplied by a factor at most $9/16$. If $j = q = 1$ then $a + 2b$ decreases, but by a factor which could be arbitrarily close to 1. The only case where $a + 2b$ does not decrease is when $j = 1$ and $q = 3$; in this case $a + 2b$ is unchanged.

This motivates us to define three classes of iterations: *good*, *bad*, and *ugly*. Let us say that we have a *good* iteration when $j \geq 2$, a *bad* iteration when $j = q = 1$, and an *ugly* iteration when $j = 1$ and $q = 3$. Since q is odd and $1 \leq q \leq 2^{j+1} - 1$, this covers all possibilities. For a bad iteration, (a, b) becomes $(b/2, a/2 + b/4)$, and for an ugly iteration, (a, b) becomes $(b/2, a/2 + 3b/4)$. We denote the matrices corresponding to good, bad and ugly iterations by G, B and U respectively. Thus

$$G = G_{j,q} = \begin{pmatrix} 0 & 1/2^j \\ 1/2^j & q/4^j \end{pmatrix},\ B = \begin{pmatrix} 0 & 1/2 \\ 1/2 & 1/4 \end{pmatrix},\ U = \begin{pmatrix} 0 & 1/2 \\ 1/2 & 3/4 \end{pmatrix}.$$

The effect of m successive ugly iterations is easily seen to be given by the matrix

$$U^m = \frac{1}{5}\begin{pmatrix} 1 + 4(-1/4)^m & 2 - 2(-1/4)^m \\ 2 - 2(-1/4)^m & 4 + (-1/4)^m \end{pmatrix}. \tag{3}$$

Assume we start from $(a, b) = (a_0, b_0)$, and after $m > 0$ successive ugly iterations we get values (a_m, b_m). Then, from Equation (3),

$$5a_m = (a + 2b) + 2(2a - b)(-1/4)^m, \tag{4}$$

$$5b_m = 2(a + 2b) - (2a - b)(-1/4)^m. \tag{5}$$

We can not have $2a_0 = b_0$ or the algorithm would have terminated. However, a_m must be an integer. This gives an upper bound on m. For a_0, b_0 of n bits, the number of successive ugly iterations is bounded by $n/2 + O(1)$ (a precise statement is made in Lemma 2).

If there were no bad iterations, this would prove that for n-bit inputs the number of iterations is $O(n^2)$, since each sequence of ugly iterations would be followed by at least one good iteration. Bad iterations can be handled by a more complicated argument which we omit, since they will be considered in detail in §2 when we discuss the complexity of the quadratic algorithm (see the proof of Theorem 2).

Since the number of iterations is $O(n^2)$ from Theorem 2, and each iteration costs time $O(M(n))$, the overall time for Algorithm CubicBinaryJacobi is $O(n^2 M(n)) = \widetilde{O}(n^3)$. Note that this worst-case bound is almost certainly too pessimistic (see §4).

2 A Provably Quadratic Algorithm

Suppose we have a sequence of $m > 0$ ugly iterations. It is possible to combine the m ugly iterations into one *harmless* iteration which is not much more expensive than a normal (good or bad) iteration. Also, it is possible to predict the maximal such m in advance. Using this trick, we reduce the number of iterations (good, bad and harmless) to $O(n)$ and their cost to $O(M(n)n) = \widetilde{O}(n^2)$.

Without loss of generality, suppose that we start from $(a_0, b_0) = (a, b)$. Since a is odd, we never have $a = 2b$.

Lemma 2. *If $\mu = \nu(a - b/2)$, then we have exactly $\lfloor \mu/2 \rfloor$ ugly iterations starting from (a, b), followed by a good iteration if μ is even, and by a bad iteration if μ is odd.*

Proof. We prove the lemma by induction on μ. If $\mu = 0$, $a - b/2$ is odd, but a is odd, so $b/2$ is even, which yields $j \geq 2$ in BinaryDividePos, thus a, b yield a good iteration. If $\mu = 1$, $a - b/2$ is even, which implies that $b/2$ is odd, thus we have $j = 1$. If we had $q = 3$ in BinaryDividePos, this would mean that $a + 3(b/2) = 0 \bmod 4$, or equivalently $a - b/2 = 0 \bmod 4$, which is incompatible with $\mu = 1$. Thus we have $q = 1$, and a bad iteration.

Now assume $\mu \geq 2$. The first iteration is ugly since 4 divides $a - b/2$, which implies that $b/2$ is odd. Thus $j = 1$, and $a - b/2 = 0 \bmod 4$

implies that $q = 3$. After one ugly iteration (a, b) becomes $(b/2, a/2 + 3b/4)$, thus $a - b/2$ becomes $-(a - b/2)/4$, and the 2-valuation of $a - b/2$ decreases by 2. □

From the above, we see that, for a sequence of m ugly iterations, $a_0, a_1, \ldots, a_m$ satisfy the three-term recurrence

$$4a_{i+1} - 3a_i - a_{i-1} = 0 \text{ for } 0 < i < m,$$

and similarly for $b_0, b_1, \ldots, b_m$. It follows that $a_i = a \bmod 4$, and similarly $b_i = b \bmod 4$, for $1 \le i < m$.

We can modify Algorithm CubicBinaryJacobi to consolidate m consecutive ugly iterations into one harmless iteration, using the expressions (4)–(5) for a_m and b_m (we give an optimised evaluation below). It remains to modify step 5 of CubicBinaryJacobi to take account of the m updates to s. Since $j = 1$ for each ugly iteration, we have to increment s by an amount

$$\delta = \sum_{0 \le i < m} \left(\frac{a_i^2 - 1}{8} + \frac{{b'_i}^2 - 1}{8} + \frac{a_i - 1}{2}\frac{b'_i - 1}{2} \right) \bmod 2,$$

where we write b'_i for $b_i/2$. However, $a_{i+1} = b'_i$ for $0 \le i < m$, so the terms involving division by 8 "collapse" mod 2, leaving just the first and last terms. The terms involving two divisions by 2 are all equal to $(a - 1)/2 \cdot (b' - 1)/2 \bmod 2$, using the observation that $a_i \bmod 4$ is constant for $0 \le i \le m$. Thus

$$\delta = \left(\frac{a_0^2 - 1}{8} + \frac{a_m^2 - 1}{8} + m\frac{a_0 - 1}{2}\frac{a_1 - 1}{2} \right) \bmod 2.$$

One further simplification is possible. Since $a_0 = a_1 \bmod 4$, and a_0 is odd, we can replace a_1 by a_0 in the last term, and use the fact that $x^2 = x \bmod 2$ to obtain

$$\delta = \left(\frac{a_0^2 - 1}{8} + \frac{a_m^2 - 1}{8} + m\frac{a_0 - 1}{2} \right) \bmod 2. \tag{6}$$

We can economise the computation of a_m and b_m from (4)–(5) by first computing

$$d = a - b', \ m = \nu(d) \text{ div } 2, \ c = (d - (-1)^m(d/4^m))/5,$$

where the divisions by 4^m and by 5 are exact; then $a_m = a - 4c$, $b_m = b + 2c$.

From these observations, it is easy to modify Algorithm CubicBinaryJacobi to obtain Algorithm QuadraticBinaryJacobi. In this algorithm, steps 7–11 implement a harmless iteration equivalent to $m > 0$ consecutive ugly iterations; steps 13–14 implement bad and good iterations, and the remaining steps are common to both. Step 5 of Algorithm CubicBinaryJacobi is split into three steps 4, 13 and 15. In the case of a harmless iteration, the computation of δ satisfying (6) is implicit in steps 4, 10 and 15.

Algorithm 2.1 QuadraticBinaryJacobi

Input: $a, b \in \mathbb{N}$ with $\nu(a) = 0 < \nu(b)$
Output: Jacobi symbol $(b|a)$

1: $s \leftarrow 0, \quad j \leftarrow \nu(b)$
2: **while** $2^j a \neq b$ **do**
3: $\quad b' \leftarrow b/2^j$
4: $\quad s \leftarrow (s + j(a^2 - 1)/8) \bmod 2$
5: $\quad (q, r) \leftarrow \text{BinaryDividePos}(a, b)$
6: $\quad$ **if** $(j, q) = (1, 3)$ **then**
7: $\quad\quad d \leftarrow a - b'$
8: $\quad\quad m \leftarrow \nu(d) \text{ div } 2$
9: $\quad\quad c \leftarrow (d - (-1)^m d/4^m)/5$
10: $\quad\quad s \leftarrow (s + m(a - 1)/2) \bmod 2$
11: $\quad\quad (a, b) \leftarrow (a - 4c, b + 2c)$ ▷ harmless iteration
12: $\quad$ **else**
13: $\quad\quad s \leftarrow (s + (a - 1)(b' - 1)/4) \bmod 2$
14: $\quad\quad (a, b) \leftarrow (b', r/2^j)$ ▷ good or bad iteration
15: $\quad s \leftarrow (s + j(a^2 - 1)/8) \bmod 2, \quad j \leftarrow \nu(b)$
16: **if** $a = 1$ **then** return $(-1)^s$ **else** return 0

Theorem 2. *Algorithm QuadraticBinaryJacobi is correct and terminates after $O(n)$ iterations of the "while" loop (steps 2–15) if the inputs are positive integers of at most n bits, with $0 = \nu(a) < \nu(b)$.*

Proof. Correctness follows from the equivalence to Algorithm CubicBinaryJacobi. To prove that convergence takes $O(n)$ iterations, we show that $a + 2b$ is multiplied by a factor at most 5/8 in each block of three iterations. This is true if the block includes at least one good iteration, so we need only consider harmless and bad iterations. Two

harmless iterations do not occur in succession, so the block must include either (harmless, bad) or (bad, bad). In the first case, the corresponding matrix is $BU^m = BU \cdot U^{m-1}$ for some $m > 0$. We saw in §1.1 that the matrix U leaves $a+2b$ unchanged, so U^{m-1} also leaves $a+2b$ unchanged, and we need only consider the effect of BU. Suppose that (a, b) is transformed into $(\widetilde{a}, \widetilde{b})$ by BU. Thus

$$\begin{pmatrix} \widetilde{a} \\ \widetilde{b} \end{pmatrix} = BU \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 1/4 & 3/8 \\ 1/8 & 7/16 \end{pmatrix} \begin{pmatrix} \widetilde{a} \\ \widetilde{b} \end{pmatrix} .$$

We see that

$$\widetilde{a} + 2\widetilde{b} = \frac{a}{2} + \frac{5b}{4} \leq \frac{5}{8}(a + 2b).$$

The case of two successive bad iterations is similar – just replace BU by B^2 in the above, and deduce that $\widetilde{a} + 2\widetilde{b} \leq (a + 2b)/2$.

We conclude that the number of iterations of the while loop is at most $cn + O(1)$, where $c = 3/\log_2(8/5) \approx 4.4243$. □

Remarks

1. A more complicated argument along similar lines can reduce the constant c to $2/\log_2(1/\rho(BU)) = 2/\log_2((11 - \sqrt{57})/2) \approx 2.5424$. Here ρ denotes the spectral radius: $\rho(A) = \lim_{k\to\infty} ||A^k||^{1/k}$.
2. In practice QuadraticBinaryJacobi is not much (if any) faster than CubicBinaryJacobi. Its advantage is simply the better worst-case time bound. A heuristic argument suggests that on average only 1/4 of the iterations of CubicBinaryJacobi are ugly.
3. Our implementations of CubicBinaryJacobi and QuadraticBinary-Jacobi are slower than GMP's $O(n^2)$ algorithm (which is based on Stein's binary gcd, as in Shallit and Sorenson [12]). However, in the next section we use the ideas of our QuadraticBinaryJacobi algorithm to get an $O(M(n) \log n)$ algorithm. We do not see how to modify the algorithm of Shallit and Sorenson to do this.[5]

3 An $O(M(n) \log n)$ Algorithm

Algorithm HalfBinaryJacobi is a modification of Algorithm Half-GB-gcd from [13]. The main differences are the following:

[5] In Algorithm Binary Jacobi in [12], it is necessary to know the sign of $a - n$ ($b - a$ in our notation) to decide whether to perform an interchange. This makes it difficult to construct a recursive $O(M(n) \log n)$ algorithm like Algorithm HalfBinaryJacobi.

1. binary division with positive (not centered) quotient is used;
2. the algorithm returns an integer s such that if a, b are the inputs, c, d the output values defined by Theorem 3, then

$$(b|a) = (-1)^s (d|c);$$

3. at steps 4 and 27, we reduce mod 2^{2k_1+2} (resp. 2^{2k_2+2}) instead of mod 2^{2k_1+1} (resp. 2^{2k_2+1}), so that we have enough information to correctly update s_0 at steps 10, 17, 21 and 25;
4. we have to "cut" some harmless iterations in two (step 15).

Remarks. The matrix Q occurring at step 19 is just $2^{2m}U^m$, where U^m is given by Equation (3). Similarly, the matrix Q occurring at step 23 is $2^{2j_0}G_{j_0,q}$. In practice, steps 13–20 can be omitted (so the algorithm becomes a fast version of CubicBinaryJacobi) – this variant is simpler and slightly faster on average.

Theorem 3. *Let a, b be the inputs of Algorithm HalfBinaryJacobi, and s, j, R the corresponding outputs. If $\binom{c}{d} = 2^{-2j} R \binom{a}{b}$, then:*

$$(b|a) = (-1)^s (d|c) \quad \textit{and} \quad \nu(2^j c) \leq k < \nu(2^j d).$$

Proof (outline). We prove the theorem by induction on the parameter k. The key ingredient is that if we reduce a, b mod 2^{2k_1+1} in step 4, then the GB sequence of a_1, b_1 matches that of a, b, for the terms computed by the recursive call at step 5. This is a consequence of [13, Lemma 7] (which also holds for binary division with positive quotient). It follows that in all the binary divisions with inputs a_i, b_i in that recursive call, a_i and $b_i/2^{j_i}$ match modulo 2^{j_i+1} the corresponding values that would be obtained from the full inputs a, b (otherwise the corresponding binary quotient q_i would be wrong). Since here we reduce a, b mod 2^{2k_1+2} instead of mod 2^{2k_1+1}, a_i and $b_i/2^{j_i}$ now match modulo 2^{j_i+2} — instead of modulo 2^{j_i+1} — the values that would be obtained from the full inputs a, b, where $2^{j_i+2} \geq 8$ since $j_i \geq 1$.

At step 10, s_0 depends only on j_0 mod 2 and a' mod 8, at step 17 it depends on m mod 2 and a' mod 4, and at step 21 on a' mod 4 and b'' mod 4. Since a' and b'' at step 21 correspond to some a_i and $b_i/2^{j_i}$, it follows that a' and b'' agree mod 8 with the values that

Algorithm 3.1 HalfBinaryJacobi

Input: $a \in \mathbb{N}, b \in \mathbb{N} \cup \{0\}$ with $0 = \nu(a) < \nu(b)$, and $k \in \mathbb{N}$
Output: two integers s, j and a 2×2 matrix R

if $\nu(b) > k$ **then** ▷ $b = 0$ is possible
    Return $0, 0, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
$k_1 \leftarrow \lfloor k/2 \rfloor$
$a_1 \leftarrow a \bmod 2^{2k_1+2}, \quad b_1 \leftarrow b \bmod 2^{2k_1+2}$
$s_1, j_1, R \leftarrow \text{HalfBinaryJacobi}(a_1, b_1, k_1)$
$a' \leftarrow 2^{-2j_1}(R_{1,1}a + R_{1,2}b), \quad b' \leftarrow 2^{-2j_1}(R_{2,1}a + R_{2,2}b)$
$j_0 \leftarrow \nu(b')$
if $j_0 + j_1 > k$ **then**
    Return s_1, j_1, R
$s_0 \leftarrow j_0({a'}^2 - 1)/8 \bmod 2$
$q, r \leftarrow \text{BinaryDividePos}(a', b')$
$b'' \leftarrow b'/2^{j_0}$
if $(j_0, q) = (1, 3)$ **then**
    $d \leftarrow a' - b''$
    $m \leftarrow \min(\nu(d) \text{ div } 2, k - j_1)$
    $c \leftarrow (d - (-1)^m d/4^m)/5$
    $s_0 \leftarrow s_0 + m(a' - 1)/2 \bmod 2$
    $(a_2, b_2) \leftarrow (a' - 4c, 2(b'' + c))$ ▷ harmless iteration
    $Q \leftarrow \begin{pmatrix} (4^m + 4(-1)^m)/5 & 2(4^m - (-1)^m)/5 \\ 2(4^m - (-1)^m)/5 & (4^{m+1} + (-1)^m)/5 \end{pmatrix}$
else
    $s_0 \leftarrow s_0 + (a' - 1)(b'' - 1)/4 \bmod 2$
    $(a_2, b_2) \leftarrow (b'', r/2^{j_0})$ ▷ good or bad iteration
    $Q \leftarrow \begin{pmatrix} 0 & 2^{j_0} \\ 2^{j_0} & q \end{pmatrix}$
    $m \leftarrow j_0$
$s_0 \leftarrow s_0 + j_0(a_2^2 - 1)/8 \bmod 2$
$k_2 \leftarrow k - (m + j_1)$
$s_2, j_2, S \leftarrow \text{HalfBinaryJacobi}(a_2 \bmod 2^{2k_2+2}, b_2 \bmod 2^{2k_2+2}, k_2)$
Return $(s_0 + s_1 + s_2) \bmod 2,\ j_1 + j_2 + m,\ S \times Q \times R$

would be computed from the full inputs, and thus the correction s_0 is correct. This proves by induction that $(b|a) = (-1)^s(d|c)$.

Now we prove that $\nu(2^j c) \leq k < \nu(2^j d)$. If there is no harmless iteration, $\nu(2^j c) \leq k < \nu(2^j d)$ is a consequence of the proof of Theorem 1 in [13]. In case there is a harmless iteration, first assume that $m = \nu(d)$ div 2 at step 15. The new values a_2, b_2 at step 18 correspond to m successive ugly iterations, which yield $j = j_1 + m \leq k$. Thus $\nu(2^j a_2) \leq k$: we did not go too far, and since we are computing the same sequence of quotients as Algorithm QuadraticBinaryJacobi, the result follows. Now if $k - j_1 < \nu(d)$ div 2, we would go too far if we performed $\nu(d)$ div 2 ugly iterations, since it would give $j_0 := \nu(d)$ div $2 > k - j_1$, thus $j := j_1 + j_0 > k$, and $\nu(2^j a_2)$ would exceed k. This is the reason why we "cut" the harmless iteration at $m = k - j_1$ (step 15). The other invariants are unchanged. □

Finally we can present our $O(M(n) \log n)$ Algorithm FastBinaryJacobi, which computes the Jacobi symbol by calling Algorithm HalfBinaryJacobi. The general structure is similar to that described in [8] for several asymptotically fast GCD algorithms.

Algorithm 3.2 FastBinaryJacobi

Input: $a, b \in \mathbb{N}$ with $0 = \nu(a) < \nu(b)$
Output: Jacobi symbol $(b|a)$

1: $s \leftarrow 0, \quad j \leftarrow \nu(b)$
2: **while** $2^j a \neq b$ **do**
3: $\quad k \leftarrow \max(\nu(b),\ \ell(b) \text{ div } 3)$ ▷ $\ell(b)$ is length of b in bits
4: $\quad s', j, R \leftarrow \text{HalfBinaryJacobi}(a, b, k)$
5: $\quad s \leftarrow (s + s') \bmod 2$
6: $\quad (a, b) \leftarrow 2^{-2j}(R_{1,1}a + R_{1,2}b, R_{2,1}a + R_{2,2}b), \quad j \leftarrow \nu(b)$
7: **if** $a = 1$ **then** return $(-1)^s$ **else** return 0

Daireaux, Maume-Deschamps and Vallée [5] prove that, for the positive binary division, the average increase of the most significant bits is 0.65 bits/iteration (which partly cancels an average decrease of two least significant bits per iteration); compare this with only 0.05 bits/iteration on average for the centered division.[6]

[6] We have computed more accurate values of these constants: 0.651993 and 0.048857 respectively.

4 Experimental Results

We have implemented the different algorithms in C (using 64-bit integers) and in GMP (using multiple-precision integers), as well as in Maple/Magma (for testing purposes).

For $\max(a, b) < 2^{26}$ the maximum number of iterations of Algorithm CubicBinaryJacobi is 64, with $a = 15548029$ and $b = 66067306$. The number of iterations seems to be $O(n)$ for $a, b < 2^n$: see Table 1. This is plausible because, from heuristic probabilistic arguments, we expect about half of the iterations to be good, and experiments confirm this. For example, if we consider all admissible $a, b < 2^{20}$, the cumulated number of iterations is 3.585×10^{12} for 2^{38} calls, i.e., an average of 13.04 iterations per call (max 48); the cumulated number of good, bad and ugly iterations is 51.78%, 25.47%, and 22.75% respectively. For $a, b < 2^{60}$, a random sample of 10^8 pairs (a, b) gave 42.72 iterations per call (max 89), with 50.54%, 25.14%, and 24.31% for good, bad and ugly respectively. These ratios seem to be converging to the heuristically expected $1/2 = 50\%$, $1/4 = 25\%$, and $1/4 = 25\%$.

When we consider all admissible $a, b < 2^{20}$, the maximum number of iterations of QuadraticBinaryJacobi is 37 when $a = 933531$, $b = 869894$, the cumulated number of iterations is 3.405×10^{12} (12.39 per call), the cumulated number of good, bad and harmless iterations is 54.51%, 26.82%, and 18.67% respectively. For $a, b < 2^{60}$, a random sample of 10^8 pairs (a, b) gave 40.21 iterations per call (max 76), with 53.70%, 26.71%, and 19.59% for good, bad and harmless respectively. These ratios seem to be converging to the heuristically expected $8/15 = 53.33\%$, $4/15 = 26.67\%$, and $1/5 = 20\%$.

We have also compared the time and average number of iterations for huge numbers, using the fast gcd algorithm in GMP, say `gcd` — which implements the algorithm from [8] — and an implementation of the algorithm from [13], say `bgcd`. For inputs of one million 64-bit words, `gcd` takes about 45.8s on a 2.83Ghz Core 2, while `bgcd` takes about 48.3s and 32,800,000 iterations: this is in accordance with the fact proven in [5] that each step of the binary gcd discards on average two least significant bits, and adds on average about 0.05 most significant bits. Our algorithm `bjacobi` (based on Algorithms 3.1–3.2) takes about 83.1s and 47,500,000 iterations (for

a version with steps 13–20 of Algorithm 3.1 omitted in the basecase routine), which agrees with the theoretical drift of 0.651993 bits per iteration. The break-even point between the $O(n^2)$ implementation of the Jacobi symbol in GMP 4.3.1 and our $O(M(n)\log n)$ implementation is about 535 words, that is about $34,240$ bits or about $10,300$ decimal digits (see Fig. 1).

5 Concluding Remarks

Weilert [15] says: "*We are not able to use a GCD calculation in* $\mathbb{Z}[i]$ *similar to the binary GCD algorithm* $\cdots$ *because we do not get a corresponding quotient sequence in an obvious manner*". In a sense we filled that gap for the computation of the Jacobi symbol, because we showed how it can be computed using a binary GCD algorithm without the need for a quotient sequence.

We showed how to compute the Jacobi symbol with an asymptotically fast time bound, using a binary GCD algorithm without the need for a quotient sequence. Our implementation is faster than a good $O(n^2)$ implementation for numbers with bitsize $n > 35000$. Our subquadratic implementation is available from `http://www.loria.fr/~zimmerma/software/#jacobi`.

Binary division with a centered quotient does not seem to give a subquadratic algorithm; however we can use it with the "cubic" algorithm (which then becomes provably quadratic) since then we control the sign of a, b. For a better quadratic algorithm, we can choose the quotient q so that $abq < 0$, by replacing q by $q - 2^{j+1}$ if necessary: experimentally, this gains on average 2.194231 bits per iteration, compared to 1.951143 for the centered quotient, and 1.348008 for the positive quotient. In comparison, Stein's "binary" algorithm gains on average 1.416488 bits per iteration [4, §7][7, §4.5.2].

n	iterations	example (a, b)	n	iterations	example (a, b)
5	6	$(7, 30)$	22	53	$(2214985, 2781506)$
10	19	$(549, 802)$	23	55	$(1383497, 8292658)$
15	34	$(23449, 19250)$	24	58	$(2236963, 12862534)$
20	48	$(656227, 352966)$	25	62	$(28662247, 30847950)$
21	51	$(1596811, 1493782)$	26	64	$(15548029, 66067306)$

Table 1. Worst cases for CubicBinaryJacobi$(b|a)$, $\max(a, b) < 2^n$.

Acknowledgement. The authors thank Steven Galbraith who asked them about the existence of an $O(M(n)\log n)$ algorithm for the Jacobi symbol, Arnold Schönhage for his comments and a pointer to the work of his former student André Weilert, and Damien Stehlé who suggested adapting the binary gcd algorithm. We also thank INRIA for its support of the ANC "équipe associée". The first author acknowledges the support of the Australian Research Council.

References

1. Eric Bach, A note on square roots in finite fields, *IEEE Trans. on Information Theory*, **36**, 6 (1990), 1494–1498.
2. Eric Bach and Jeffrey O. Shallit, *Algorithmic Number Theory, Volume 1: Efficient Algorithms*, MIT Press, 1996. Solution to problem 5.52.
3. Paul Bachmann, *Niedere Zahlentheorie*, Vol. 1, Teubner, Leipzig, 1902. Reprinted by Chelsea, New York, 1968.
4. Richard P. Brent, Twenty years' analysis of the binary Euclidean algorithm, in *Millennial Perspectives in Computer Science: Proceedings of the 1999 Oxford – Microsoft Symposium in honour of Professor Sir Antony Hoare* (edited by J. Davies, A. W. Roscoe and J. Woodcock), Palgrave, New York, 2000, 41–53. `http://wwwmaths.anu.edu.au/~brent/pub/pub183.html`
5. Benoît Daireaux, Véronique Maume-Deschamps and Brigitte Vallée, The Lyapunov tortoise and the dyadic hare, *Proceedings of the 2005 International Conference on Analysis of Algorithms*, DMTCS Proc. AD (2005), 71–94. `http://www.dmtcs.org/dmtcs-ojs/index.php/proceedings/issue/view/81`
6. Carl F. Gauss, Neue Beweise und Erweiterungen des Fundamentalsatzes in der Lehre von den quadratischen Resten, reprinted in *Untersuchungen über Höhere Arithmetik*, Chelsea, New York, 1965, page 509.
7. Donald E. Knuth, *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*, third edition, Addison-Wesley, 1997.
8. Niels Möller, On Schönhage's algorithm and subquadratic integer GCD computation, *Mathematics of Computation* **77**, 261 (2008), 589–607.
9. Arnold Schönhage, Schnelle Berechnung von Kettenbruchentwicklungen, *Acta Informatica* **1** (1971), 139–144.
10. Arnold Schönhage, personal communication by email, December 2009.
11. Arnold Schönhage, Andreas F. W. Grotefeld and Ekkehart Vetter, *Fast Algorithms: A Multitape Turing Machine Implementation*, BI-Wissenschaftsverlag, Mannheim, 1994.
12. Jeffrey Shallit and Jonathan Sorenson, A binary algorithm for the Jacobi symbol, *ACM SIGSAM Bulletin* **27**, 1 (January 1993), 4–11. `http://euclid.butler.edu/~sorenson/papers/binjac.ps`
13. Damien Stehlé and Paul Zimmermann, A binary recursive gcd algorithm, *Proc. Sixth International Symposium on Algorithmic Number Theory*, *Lecture Notes in Computer Science* **3076** (2004), 411–425.
14. Brigitte Vallée, A unifying framework for the analysis of a class of Euclidean algorithms, *Proceedings of the LATIN'00 Conference*, *Lecture Notes in Computer Science* **1776** (2000), 343–354.
15. André Weilert, Fast Computation of the Biquadratic Residue Symbol, *Journal of Number Theory* **96** (2002), 133–151.

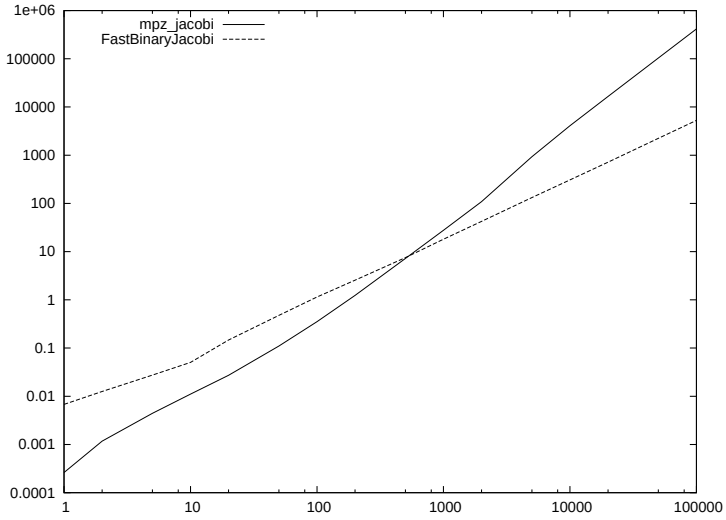

Fig. 1. Comparison of GMP 4.3.1 `mpz_jacobi` routine with our `FastBinaryJacobi` implementation in log-log scale. The x-axis is in 64-bit words, the y-axis in milliseconds on a 2.83Ghz Core 2.