Efficient computation of the Jacobi symbol

Niels Möller

January 2010

(FIXME: Possible targets: ANTS X (2012, where?), ARITH 20 (2011? Where?), some journal accepting brief contributions/letters?)

Abstract

The family of left-to-right GCD algorithms reduces input numbers by repeatedly subtracting the smaller number, or multiple of the smaller number, from the larger number. This paper describes how to extend any such algorithm to compute the Jacobi symbol, using a single table lookup per reduction. For both quadratic time GCD algorithms (Euclid, Lehmer) and subquadratic algorithms (Knuth, Schönhage, Möller), the additional cost is linear, roughly one table lookup per quotient in the quotient sequence.

1 Introduction

The Legendre symbol and its generalizations, the Jacobi symbol and the Kronecker symbol, are important functions in number theory. **(FIXME: Say something more interesting?)**. For simplicity, in this paper we focus on computation of the Jacobi symbol, since the Kronecker symbol can be computed by the same function with a little preprocessing of the inputs.

1.1 Jacobi and gcd

Two quadratic algorithms for computing the Kronecker symbol (and hence also the Jacobi symbol) are described as Algorithm 1.4.10 and 1.4.12 in [3]. These algorithms run in quadratic time, and consists of a series of reduction steps, related to Euclid's GCD algorithm and the binary GCD algorithm, respectively. Both Kronecker algorithms share one property with the binary GCD algorithm: The reduction steps examine the current pair of numbers in both ends. They examine the least significant end to cast out powers of two, and they examine the most significant end to determine a quotient (like in Euclid's algorithm) or to determine which number is largest (like in the binary GCD algorithm).

Fast, subquadratic, GCD algorithms work by divide-and-conquer, where a substantial piece of the work is done by examining only one half of the input numbers. Fast left-to-right GCD is related to fast algorithms for computing the continued fraction expansion [7, 5]. These are left-to-right algorithms, in that they process the input from the most significant end. The binary recursive algorithm [9] is a right-to-left algorithm, in that it processes inputs from the least significant end. The asymptotic running times of these algorithms are $O(M(n) \log n)$, where $M(n)$ denotes the time needed to multiply two n-bit numbers. The GCD algorithm used in recent versions of the GMP library [4] is a variant of Schönhage's algorithm [6].

It is possible to compute the Jacobi symbol in subquadratic time, with the same asymptotic complexity as GCD. One algorithm is described in [1] (solution to exercise 5.52), which says:

> This complexity bound is part of the "folklore" and has apparently never appeared in print. The basic idea can be found in Gauss [1876]. Our presentation is based on that in Bachmann [1902]. H. W. Lenstra, Jr. also informed us of this idea; he at-

tributes it to A. Schönhage.

Since the quadratic algorithms for the Jacobi symbol examines the data at both ends, some reorganization is necessary to construct a divide-and-conquer algorithm that processes data from one end. The binary GCD algorithm has the same problem. In the binary recursive GCD algorithm, this is handled by using a slightly different reduction step using 2-adic division.

Recently, the binary recursive GCD algorithm has been extended to compute the Jacobi symbol [2]. The main difference to the corresponding GCD algorithm is that it needs the intermediate reduced values to be non-negative, and to ensure this the binary quotients must be chosen in the range $1 \leq q < 2^{k+1}$ rather than $|q| < 2^k$. As a result, the algorithm is slower than the GCD algorithm by a small constant factor.

1.2 Main contribution

This paper describes a fairly simple extension to a wide class of left-to-right GCD algorithms, including Lehmer's algorithm and the subquadratic algorithm in [6], which computes the Jacobi symbol using only $O(n)$ extra time and $O(1)$ extra space[1]. This indicates that also for the fastest algorithms for large inputs, the cost is essentially the same for computing the GCD and computing the Jacobi symbol.[2]

Like the algorithm described in [1], the computation is related to the quotient sequence. The updates of the Jacobi symbol are somewhat different, instead following an unpublished algorithm by Schönhage [8] for computing the Jacobi symbol from the quotient sequence modulo four. In the GCD algorithms in GMP, the quotients are not always applied in a single step; instead, there is a series of reductions of the form $a \leftarrow a - mb$, where m is a positive number equal to or less than the correct quotient $\lfloor a/b \rfloor$. In the corresponding Jacobi algorithms, the Jacobi sign is updated for each such partial quotient. Most of the partial quotients are determined from truncated inputs where the least significant parts of the numbers are ignored. The least significant two bits, needed for the Jacobi computation, must therefore be maintained separately.

[1]The size of the additional state to be maintained is $O(1)$. But in a practical implementation, which does not store this state in a global variable, either the state or a pointer to it will be copied into each activation record, which for a subquadratic recursive divide-and-conquer algorithm costs $O(\log n)$ extra space rather than $O(1)$

[2]Even though we cannot rule out the existence of a left-to-right GCD algorithm which is a constant factor faster than Jacobi. Such an algorithm would lie outside the class of "generic left-to-right GCD algorithms" we describe in this paper, e.g., it might use intermediate reduced values of varying signs and quotients that are rounded towards the nearest integer rather than towards $-\infty$.

1.3 Notation

The time needed to multiply two n-bit numbers is denoted $M(n)$, where $M(n) = O(n \log n 2^{\log^* n})$ for the fastest known algorithms. **(FIXME: Should we refer to some or all of Schönhage-Strassen, Fürer and "Anindya De, Piyush P Kurur, Chandan Saha, Ramprasad Saptharishi. Fast Integer Multiplication Using Modular Arithmetic. Symposium on Theory of Computation (STOC) 2008."?)**

The Jacobi symbol is denoted $(a|b)$. We use the convention that [condition] means the function that is one when the condition is true, otherwise 0, e.g., $(0|b) = [b = 1]$. **(FIXME: Refer to Concrete Mathematics?)**

2 Left-to-right gcd algorithms

In this paper, we will not describe the details of fast GCD algorithms. Instead we will consider Algorithm 1, which is a generic left-to-right GCD algorithm, with a basic reduction step where a multiple of the smaller number is subtracted from the larger number. We also describe the main idea of fast instantiations of this algorithm.

This algorithm terminates after a finite number of steps, since in each iteration $\max(a, b)$ is reduced, until $a = b$ and the algorithm terminates. It returns the correct value, since $\mathrm{GCD}(a, b)$ is unchanged by each reduction step.

The running time of an instantiation of this algorithm depends on the choice of m in each step, and on

```
$g \leftarrow \text{GCD}(a, b)$
    In: $a, b > 0$
1   repeat
2       if $a \geq b$
3           $a \leftarrow a - mb$, with $1 \leq m \leq \lfloor a/b \rfloor$
4           if $a = 0$
5               return $b$
6       else
7           $b \leftarrow b - ma$, with $1 \leq m \leq \lfloor b/a \rfloor$
8           if $b = 0$
9               return $a$
```

Algorithm 1: Generic left-to-right GCD algorithm.

the amount of computation done in each step. E.g., if $m = 1$, the worst case number of iterations in exponential. Euclid's algorithm is a special case where, in each step, m is the correct quotient of the current numbers.

The faster algorithms implements an iteration that depends only on some of the most significant bits of a and b: These bits determine which of a and b is largest, and they also suffice for computing an m which is close to the quotient $\lfloor a/b \rfloor$ or $\lfloor b/a \rfloor$. Furthermore, one can compute an initial part of the sequence of reductions based on the most significant parts of a and b, collect the reductions into a transformation matrix, and apply all the reductions at once to the least significant parts of a and b later on. This saves a lot of time, since it omits computing all the intermediate a and b to full precision. If one repeatedly chops off one or two of the most significant words, one gets Lehmer's algorithm, and by chopping numbers in half, one can construct a divide-and-conquer algorithm with subquadratic complexity.

We will extend this generic algorithm to also compute the Jacobi symbol. To do that, we need to investigate how the basic reduction $a - mb$ affects the Jacobi symbol. When we have sorted this out, in the next section, the result is easily applied to all variants of Algorithm 1.

3 Left-to-right Jacobi

In this section, we summarize the properties of the Jacobi symbol we use, derive the update rules needed for our left-to-right algorithm. Finally, we give the resulting algorithm and prove its correctness.

3.1 Jacobi symbol properties

The Jacobi symbol $(a|b)$ is defined for b odd and positive, and arbitrary a. We work primarily with nonnegative a, and make use of the following properties of the Jacobi symbol.

Proposition 1 *Assume that a is positive and that b is odd and positive. Then*

(i) $(0|b) = [b = 1]$.

(ii) $(a|b) = (-1)^{(b-1)/2}(-a|b)$

(iii) If both a and b are odd, then

$$(a|b) = (-1)^{(a-1)(b-1)/4}(b|a)$$

(iv) $(a|b) = (a - mb|b)$ *for any m.*

(v) If $a = 0 \pmod 4$ and $1 \leq m \leq \lfloor b/a \rfloor$, then

$$(a|b) = (a|b - ma)$$

(vi) If $a = 2 \pmod 4$ and $1 \leq m \leq \lfloor b/a \rfloor$, then

$$(a|b) = (-1)^{m(b-1)/2 + m(m-1)/2}(a|b - ma)$$

Proof: For (i) to (iv) we refer to standard textbooks. The final two are not so well-known, and their use for Jacobi computation is suggested by Schönhage [8]. To prove them, first note that

$$\begin{aligned}
(a|b) &= (a - b|b) && \text{By (iv)} \\
&= (-1)^{(b-1)/2}(b - a|b) && \text{By (ii)} \\
&= (-1)^{(b-1)/2 + (b-1)(b-a-1)/4}(b|b - a) && \text{By (iii)} \\
&= (-1)^{(b-1)/2 + (b-1)(b-a-1)/4}(a|b - a) && \text{By (iv)}
\end{aligned}$$

Since $b^2 - 1$ is divisible by 8 for any odd b, we get a resulting exponent, modulo two, of

$$(b-1)/2 + (b-1)(b-a-1)/4 = a(b-1)/4$$

If $a = 0 \pmod 4$, this exponent is even and hence there is no sign change. And this continues to hold if the subtraction is repeated, which proves (v). Next, consider the case $a = 2 \pmod 4$. Then $a/2 = 1 \pmod 2$, and repeating the subtraction m times gives the exponent

$$\begin{aligned} a\{(b-1)+(b-a-1)+\cdots+(b-(m-1)a-1)\}/4 \\ = m(b-1)/2 + m(m-1)/2 \pmod 2 \end{aligned}$$

which proves (vi). □

Finally, note that in these formulas, all the signs are determined by the least significant two bits of a, b and m.

3.2 The new algorithm

The GCD algorithm works with two non-negative integers a and b, where multiples of the smaller one is subtracted from the larger. To compute the Jacobi symbol we maintain these additional state variables:

$e \in \mathbb{Z}_2$	Current sign is $(-1)^e$
$\alpha \in \mathbb{Z}_4$	Least significant bits of a
$\beta \in \mathbb{Z}_4$	Least significant bits of b
$d \in \mathbb{Z}_2$	Index of denominator

(FIXME: Use indices instead of α, β, to make the algorithm description more concise? Introduce S for the state variables taken together.)

The value of d is one if the most recent reduction subtracted b from a, and zero if it subtracted a from b. The state is updated by the function JUPDATE, Algorithm 2. Since the inputs of this function are nine bits, and the outputs are six bits, it's clear it can be implemented using a lookup table consisting of 2^9 six-bit entries, which fits in 512 bytes if entries are padded to byte boundaries.[3]

Algorithm 3 extends the generic left-to-right GCD algorithm to compute the Jacobi symbol. The main loop of this algorithm differs from Algorithm 1 only by the calls to JUPDATE for each reduction step.

[3]One quarter of the entries in this table corresponds to invalid inputs, since at least one of α and β is always odd. If we also note that the value of d is needed only when $\alpha = \beta = 3$, the state can be encoded into only 26 values, and then the table can be compacted to only 208 entries.

```
$S \leftarrow$ JUPDATE$(S, d_{\text{new}}, m)$
    In: $d_{\text{new}} \in \mathbb{Z}_2$, $m \in \mathbb{Z}_4$
    State: $S = (e, \alpha, \beta, d)$
    **if** $d \neq d_{\text{new}}$ and both $\alpha$ and $\beta$ are odd
        $e \leftarrow e + (\alpha - 1)(\beta - 1)/4$        // Reciprocity
    $d \leftarrow d_{\text{new}}$
    **if** $d = 1$
        **if** $\beta = 2$
            $e \leftarrow e + m(\alpha - 1)/2 + m(m-1)/2$
        $\alpha \leftarrow \alpha - m\beta$
    **else**
        **if** $\alpha = 2$
            $e \leftarrow e + m(\beta - 1)/2 + m(m-1)/2$
        $\beta \leftarrow \beta - m\alpha$
    **return** $(e, \alpha, \beta, d)$
```

Algorithm 2: Updating the state of the Jacobi symbol computation.

```
$j \leftarrow$ JACOBI$(a, b)$
    In: $a, b > 0$, $b$ odd
    Out: The Jacobi symbol $(a|b)$
    State: $S = (e, \alpha, \beta, d)$
    $S \leftarrow (0, a \bmod 4, b \bmod 4, 1)$
    **repeat**
        **if** $a \geq b$
            $a \leftarrow a - mb$, with $1 \leq m \leq \lfloor a/b \rfloor$
            $S \leftarrow$ JUPDATE$(S, 1, m \bmod 4)$
            **if** $a = 0$
                **return** $[b = 1](-1)^e$
        **else**
            $b \leftarrow b - ma$, with $1 \leq m \leq \lfloor b/a \rfloor$
            $S \leftarrow$ JUPDATE$(S, 0, m \bmod 4)$
            **if** $b = 0$
                **return** $[a = 1](-1)^e$
```

Algorithm 3: The algorithm for computing the Jacobi symbol. **(FIXME: Better choice of letter for the return value? Return both gcd and Jacobi symbol?)**

3.3 Correctness

Let a_0 and b_0 denote the original inputs to Algorithm 3. Since the reduction steps and the stop condition are the same as in Algorithm 1, it terminates after a finite number of steps. We now prove that it returns $(a_0|b_0)$.

Algorithm 3 clearly maintains $\alpha = a \bmod 4$ and $\beta = b \bmod 4$. We next prove that the following holds at the start of each iteration:

If $d = 0$ we have

$$(a_0|b_0) = (-1)^e \times \begin{cases} (b|a) & \alpha \text{ odd} \\ (a|b) & \alpha \text{ even} \end{cases} \tag{1}$$

and if $d = 1$ we have

$$(a_0|b_0) = (-1)^e \times \begin{cases} (a|b) & \beta \text{ odd} \\ (b|a) & \beta \text{ even} \end{cases} \tag{2}$$

This clearly holds at the start of the loop, to prove that it is maintained, consider the case $a \geq b$ (the case $a < b$ is analogous). Let a, b (unchanged) and $S = (e, \alpha, \beta, d)$ denote the values of the variables before line 4. There are a couple of different cases, depending on the state:

- If β is odd and either α is even or $d = 1$, then $(a_0|b_0) = (-1)^e(a|b) = (-1)^e(a - mb|b)$.
- If α and β are both odd and $d = 0$, then $(a_0|b_0) = (-1)^e(b|a) = (-1)^{e+(a-1)(b-1)/4}(a - mb|b)$.
- If $\beta = 0 \pmod 4$, then $(a_0|b_0) = (-1)^e(b|a) = (-1)^e(b|a - mb)$.
- If $\beta = 2 \pmod 4$, then $(a_0|b_0) = (-1)^e(b|a) = (-1)^{e+m(a-1)/2+m(m-1)/2}(b|a - mb)$.

In each case, the call to JUPDATE makes the appropriate change to e, and Eq. (2) holds after the iteration.

4 Results

(FIXME: I guess we need to say something quantitative about implementation and benchmarks?)

Acknowledgements

The author wishes to thank Richard Brent for providing valuable background material and for encouraging the writing of this paper.

References

[1] Eric Bach and Jeffrey Shallit. *Algorithmic Number Theory, Vol. 1: Efficient Algorithms.* Foundations of Computing. MIT Press, 1996.

[2] Richard P. Brent and Paul Zimmermann. An O(M(n) log n) algorithm for the Jacobi symbol. *Submitted paper*, Januari 2010. Preprint at `http://wwwmaths.anu.edu.au/~brent/pub/pub236.html`.

[3] Henri Cohen. *A course in computational algebraic number theory.* Springer, 1996.

[4] Torbjörn Granlund. GNU multiple precision arithmetic library, version 4.3, May 2009. `http://gmplib.org/`.

[5] Donald E. Knuth. The analysis of algorithms. *Actes du Congrés International des Mathématiciens*, pages 269–274, 1970.

[6] Niels Möller. On Schönhage's algorithm and subquadratic integer gcd computation. *Mathematics of Computation*, 2008.

[7] Arnold Schönhage. Schnelle Berechnung von Kettenbruchentwicklungen. *Acta Informatica*, 1:139–144, 1971.

[8] Arnold Schönhage. Email to Richard P. Brent, 2009. Excerpt from 1987 notes.

[9] Damien Stehlé and Paul Zimmermann. A binary recursive GCD algorithm. In D. Buell, editor, *ANTS-VI*, volume 3076 of *LCNS*, Burlington, June 2004. Springer-Verlag.