Computing the Jacobi symbol using Bernstein-Yang

Mike Hamburg

Rambus, Inc. mhamburg@rambus.com

Abstract. Number-theoretic algorithms often need to calculate one or both of two related quantities: *modular inversion* and *Jacobi symbol*. These two functions seem unrelated at first glance, but in fact the algorithms for calculating them are closely related: they can both be calculated either by variants of Euclid's GCD algorithm, or when the modulus is prime, by exponentiation. As a result, an implementation of one algorithm can often be adapted to compute the other instead, or they can even be calculated together in a batch.

The Bernstein-Yang right-to-left modular inversion algorithm is notable for taking constant, asymptotically subquadratic time. Right-to-left algorithms are tricky to adapt for the Jacobi symbol, because they do not consider the signs of the values being operated on. But the Jacobi symbol is defined only on positive integers, and the rules for computing it need corrections if negative integers are introduced.

In this short paper, we show how to overcome this difficulty and produce a right-to-left Jacobi symbol algorithm based on Bernstein-Yang.

Keywords: Jacobi symbol, modular inversion, Bernstein-Yang algorithm, extended Euclidean algorithm

1 Introduction

The modular inversion function $x^{-1} \bmod y$ is frequently used in public-key cryptography, as is the Jacobi symbol calculation $\left(\frac{x}{y}\right)$. The two functions are used especially often in elliptic curve cryptography: inversion is used to divide through by a projective coordinate Z at the end of the computation, or for ECDSA scalar calculations; and the Jacobi symbol is used to determine whether an x-coordinate corresponds to a point on the curve or on its quadratic twist. They can even be called simultaneously on the same value, in order to produce a final output from an x-only scalar multiplication, and simultaneously to check whether it is on the curve [Ham20].

Inversion and the Jacobi symbol can both be computed with exponentiation when the modulus is prime. They can also be computed using variants of Euclid's GCD algorithm. Previous work shows how to do this using left-to-right variants of Euclid's algorithm [Mö19] or right-to-left algorithms which avoid negative

numbers [BZ10]. This gives Jacobi symbol algorithms which run in subquadratic time in $\log y$.

The Bernstein-Yang algorithm [BY19] calculates modular inversion using a right-to-left variant of Euclid's GCD algorithm. It can be used to invert integers or polynomials; here we are interested in the integer case. Bernstein-Yang is simple, asymptotically subquadratic, and very fast in practice. It is also straightforward to implement in constant time, i.e. taking an amount of time which depends on the sizes of the inputs but not their values. These properties make Bernstein-Yang an excellent choice for implementations of inversion. However, since it is a right-to-left algorithm, it does not prevent its intermediates from becoming negative. This raises an obstacle to using it for the Jacobi symbol, which is defined only for positive moduli.

Here we show how to overcome that obstacle. This gives a simple subquadratic Jacobi symbol algorithm. Our technique might also generalize to other right-to-left Euclidean algorithms.

2 Legendre, Jacobi and Kronecker symbols

The Legendre, Jacobi and Kronecker symbols "of x on y" are all commonly written $\left(\frac{x}{y}\right)$. These symbols all take the same value where they are defined, but apply to increasingly general sets of integers (x, y). We are interested in the Jacobi symbol, but we will briefly describe all three, disambiguated as $\left(\frac{x}{y}\right)_L$, $\left(\frac{x}{y}\right)_J$ and $\left(\frac{x}{y}\right)_K$ respectively.

The Legendre symbol $\left(\frac{x}{y}\right)_L$, where x is an integer and y is an odd prime number, is defined as 0 if y divides x; or 1 if x is a quadratic residue mod y (meaning that $x \equiv z^2 \mod y$ for some integer z); or -1 if x is a quadratic nonresidue mod y.

The Jacobi symbol $\left(\frac{x}{y}\right)_J$ extends the Legendre symbol to odd positive integers y. If $y = \prod_i p_i^{e_i}$ with p_i prime, then

$$\left(\frac{x}{y}\right)_J := \prod_i \left(\frac{x}{p_i}\right)_L^{e_i}.$$

The Kronecker symbol further extends this to nonzero integers y by defining

$$\left(\frac{x}{-1}\right)_K := \begin{cases} 1 \text{ if } x \geqslant 0 \\ -1 \text{ if } x < 0 \end{cases} \quad \text{and} \quad \left(\frac{x}{2}\right)_K := \begin{cases} 0 \text{ if } x \text{ is even} & \\ 1 \text{ if } x \in \{\pm 1\} & \mod 8 \\ -1 \text{ if } x \in \{\pm 3\} & \mod 8 \end{cases}$$

and most other values multiplicatively. In the rest of this work, we will use $\left(\frac{x}{y}\right)$ to refer to the Jacobi symbol. It obeys the relations:

$$\left(\frac{x+ky}{y}\right) = \left(\frac{x}{y}\right) \tag{1}$$

$$\left(\frac{xx'}{y}\right) = \left(\frac{x}{y}\right) \cdot \left(\frac{x'}{y}\right) \tag{2}$$

$$\left(\frac{-1}{y}\right) = \begin{cases} -1 \text{ if } y \equiv 3 \mod 4 \\ 1 \text{ otherwise} \end{cases} \tag{3}$$

$$\left(\frac{2}{y}\right) = \begin{cases} -1 \text{ if } y \in \{3,5\} \mod 8 \\ 1 \text{ otherwise} \end{cases} \tag{4}$$

$$\left(\frac{y}{x}\right) = \left(\frac{x}{y}\right) \cdot \begin{cases} -1 \text{ if } x \equiv y \equiv 3 \mod 4 \\ 1 \text{ otherwise} \end{cases} \tag{5}$$

$$\left(\frac{1}{y}\right) = \left(\frac{x}{1}\right) = 1 \qquad \text{and} \qquad \left(\frac{0}{y}\right) = 0 \text{ for } y \neq 1 \tag{6}$$

We will need to extend the Jacobi symbol to negative odd y. We could use the Kronecker symbol, but it inconveniently doesn't obey (1) or even (2) in all cases. So instead we will use the Jacobi symbol $\left(\frac{x}{|y|}\right)$; this behaves similarly to the Kronecker symbol, but is more convenient for our algorithm. Let

$$s_{x,y} := \begin{cases} -1 \text{ if } x < 0 \text{ and } y < 0 \\ 1 \text{ otherwise.} \end{cases}$$

Then $\left(\frac{x}{|y|}\right)$ obeys the same relations as $\left(\frac{x}{y}\right)$, except that:

$$\left(\frac{-1}{|y|}\right) = s_{-1,y} \cdot \begin{cases} -1 \text{ if } y \equiv 3 \mod 4 \\ 1 \text{ otherwise} \end{cases} \tag{3'}$$

$$\left(\frac{y}{|x|}\right) = \left(\frac{x}{|y|}\right) \cdot s_{x,y} \cdot \begin{cases} -1 \text{ if } x \equiv y \equiv 3 \mod 4 \\ 1 \text{ otherwise} \end{cases} \tag{5'}$$

By negating x first, we can turn (5′) into a "90° rotation rule":

$$\left(\frac{x}{|y|}\right) = \left(\frac{-y}{|x|}\right) \cdot s_{x,-y} \cdot \begin{cases} -1 \text{ if } x \equiv -y \equiv 3 \mod 4 \\ 1 \text{ otherwise} \end{cases} \tag{5''}$$

This will be our main quadratic reciprocity rule, because Bernstein-Yang rotates (x, y) to $(-y, x)$ instead of swapping them.

3 Euclidean algorithms

The most common algorithms for computing inversion and Jacobi symbols are closely related. When y is prime, then $x^{-1} \equiv x^{y-2} \mod y$, and $\left(\frac{x}{y}\right) = x^{(y-1)/2}$

mod y. However, these approaches take cubic time to compute with the $O(\log^2 y)$ schoolbook multiplication algorithm, and more than quadratic time with faster multiplication algorithms. Furthermore, they do not work for composite y.

A faster and more general approach is to use an extension of Euclid's greatest common divisor (gcd) algorithm, which reduces

$$\gcd(x, y) = \gcd(y, x) = \gcd(y \bmod x, x).$$

The Euclidean algorithm can be extended to compute modular inversion by tracking a matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ such that

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}.$$

The matrix can be computed by writing $y \bmod x = y - kx$ for some k, so that the new state is a linear combination of (x, y). The algorithm ends in the state $(x', y') = (0, \gcd(x, y))$. If $y' = 1$ then we have $1 = y' = cx + dy$, so that $c = x^{-1} \bmod y$. If $y' \neq 1$ then $x^{-1} \bmod y$ is not defined. Because only c is needed but not b or d, only the first column of this matrix needs to be calculated.

The *binary Euclidean algorithm* [Ste67] avoids the complexity of modular reductions. It assumes that the given y is odd, which is often the case in cryptography; if y is not odd, the algorithm first divides out powers of 2 from y and/or x until y is odd. From that point on, the binary Euclidean algorithm maintains that y is odd, by applying the relations:

$$\gcd(x, y) = \begin{cases} \gcd(x/2, y) & \text{if } x \text{ is even} \\ \gcd((y - x)/2, x) & \text{if } x \text{ is odd and } x < y \\ \gcd((x - y)/2, y) & \text{if } x \text{ is odd and } x \geqslant y \end{cases}$$

For the binary Euclidean algorithm and its variants, the entries in the tracking matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ aren't integers: after i steps they are of the form $e/2^i$ where e is an integer. This isn't a problem for inversion if y is odd, because division by $2^i \bmod y$ is simply Montgomery reduction, which is a simple and efficient operation.

Because y' is positive and odd throughout the binary Euclidean algorithm, the value $\left(\frac{x'}{y'}\right)$ is always defined. The algorithm can be extended by using the relations (2)-(6) to track a sign s such that $\left(\frac{x}{y}\right) = s \cdot \left(\frac{x'}{y'}\right)$. In the ending state, if $y' = 1$ then $\left(\frac{x'}{y'}\right) = 1$, so $s = \left(\frac{x}{y}\right)$; otherwise $\left(\frac{x}{y}\right) = 0$.

3.1 Bernstein-Yang

The Bernstein-Yang algorithm [BY19] and its variants replace the numerical comparisons $x \overset{?}{<} y$ in the binary GCD algorithm by an estimator $\delta \approx \log_2(y/x)$.

The value δ starts at 1, and the input y is assumed to be odd. The algorithm uses the update step:

$$(\delta', x', y') = \begin{cases} (1-\delta,\ (x-y)/2,\ x) & \text{if } \delta > 0 \text{ and } x \text{ is odd} \\ (1+\delta,\ (x+y)/2,\ y) & \text{if } \delta \leqslant 0 \text{ and } x \text{ is odd} \\ (1+\delta,\ x/2,\ y) & \text{if } x \text{ is even} \end{cases}$$

Variants of Bernstein-Yang have the same flow, but slightly different handling of the δ terms [HAS21,WMt]. These algorithms can be shown to reach $(0, \gcd(x, y))$ in at most $O(\max(\log x, \log y))$ steps, with concrete constants. By running the algorithm for that maximum number of steps, Bernstein-Yang becomes a "constant-time" algorithm. That is, its runtime depends on bounds of the bit-lengths of the inputs (e.g. $x < y = 2^{255} - 19$), but not on the actual values of (x, y).

A major advantage of this approach is that a sequence of k steps of the algorithm depend only on δ and on the least-significant k bits of x and y. This means that Bernstein-Yang can be performed in batches. For some batch size k, one can determine an update matrix $\begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix}$ based on the least-significant k bits of (x, y) and the current value of δ, and then one can apply this to to the state[1] $\begin{pmatrix} a & b & x \\ c & d & y \end{pmatrix}$. This can be done recursively: $\begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix}$ can be calculated using sub-batches of size $k' < k$ to compute a step matrix $\begin{pmatrix} a'' & b'' \\ c'' & d'' \end{pmatrix}$, etc. This recursion gives Bernstein-Yang an asymptotically sub-quadratic runtime, if sub-quadratic multiplication (e.g. Karatsuba or FFT multiplication) is used.

4 Tracking the Jacobi symbol in Bernstein-Yang

In this section, we describe the sign of a number as either "negative" or "non-negative", so that there are only two possibilities. This definition accords with a two's complement representation, so it's easy to determine in software or hardware.

We would like to track the Jacobi symbol through Bernstein-Yang, again by tracking a variable $s \in \{\pm 1\}$ such that $\left(\frac{x}{|y|}\right) = s \cdot \left(\frac{x'}{|y'|}\right)$. We can try to do this by applying rules (1) through (6), but we run into a problem on the recursive steps. First, rules (4) and (5″) need (x, y) mod 4 or mod 8 instead of merely mod 2. This is easy to solve: to perform k steps including the Jacobi symbol, we will need $k + 2$ bits of x and y, instead of only k bits.

More importantly, because the recursive steps use only the least-significant bits of x and y, they cannot determine the sign $s_{-x,y}$ for rule (5″). This rule is used in the update step

$$(\delta', x', y') = (1 - \delta, (x - y)/2, x)$$

[1] Note that when computing inversion, only (a, c, x, y) are needed in the outermost state; and when computing GCD and/or Jacobi symbol, only (x, y) are needed.

In this case, the term $s_{x,-y}$ is -1 precisely when $y' = x < 0 < y$, i.e. when y becomes negative. This update step is the only way that y can change sign (or indeed, change at all).

We can track the product of $s_{x,-y}$ terms by observing that the update matrices rotate the state in a net counterclockwise manner, and y only changes by being rotated counterclockwise in this way. To define this formally, we will say that a matrix $M := \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ is a *ratchet matrix* if M has positive determinant and c and d are both nonnegative.

We then have the following theorem:

Theorem 1. *Let a sequence of matrices* $T_i := \begin{pmatrix} a_i & b_i \\ c_i & d_i \end{pmatrix}$ *be defined by*

$$T_0 := \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad \text{and} \qquad T_i := M_i \cdot T_{i-1} \text{ for } i > 0,$$

where the M_i *are ratchet matrices. Let* $v_i := \begin{pmatrix} x_i \\ y_i \end{pmatrix} := T_i \cdot \begin{pmatrix} x \\ y \end{pmatrix}$*, where either* $y \neq 0$ *or* $x > 0$*. Let* t_j *and* u_j *be the number of times that* y_i *and* c_i *change signs, respectively, in this sequence for* $i \leqslant j$*.*

Then $0 \leqslant t_i - u_i \leqslant 1$*. Thus* $t_i - u_i$ *is equal to its parity, meaning that*

$$t_i - u_i = \begin{cases} 1 \text{ if } (y_i < 0 \leqslant y_0 \text{ or } y_0 < 0 \leqslant y_i) \text{ xor } c_i < 0 \\ 0 \text{ otherwise.} \end{cases}$$

Proof. We will first rewrite the sequence to simplify it. If a ratchet matrix $M_i = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ has $c = 0$, then it certainly has $d > 0$ and $a > 0$ because $d \geqslant 0$ and the determinant $ad > 0$. Otherwise, it has $c > 0$, and can be factored as

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} 1 & a \\ 0 & c \end{pmatrix} \cdot \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & d/c \\ 0 & ad/c - b \end{pmatrix}$$

where $ad/c - b = |M|/c > 0$. All three of these matrices are ratchet, and the first and last of them do not change the signs of c_i or y_i. So by expanding matrices in this way we may assume that all the M_i are either a skewing matrix of the form $\begin{pmatrix} a & b \\ 0 & d \end{pmatrix}$ with $(a, b) > 0$, or are the 90° rotation $\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$.

Next, let $z_i := (T_i)_{00}$. Because each $|M_i| > 0$, certainly $|T_i| > 0$, so that

$$\begin{vmatrix} x_0 & 1 \\ y_0 & 0 \end{vmatrix} = -y_0 \qquad \text{and} \qquad \left| T_i \begin{pmatrix} x_0 & 1 \\ y_0 & 0 \end{pmatrix} \right| = \begin{vmatrix} x_i & a_i \\ y_i & c_i \end{vmatrix} = x_i c_i - y_i a_i$$

have the same sign.

The proof concludes by induction on i, using case analysis. The base case $i = 0$ is trivial, and the induction step for skewing matrices is trivial because neither y_i nor c_i can change sign, so we only need to analyze the rotation case where $(y_{i+1}, c_{i+1}) = (x_i, a_i)$, and either y_i or c_i changes sign.

We first analyze the case that $y_0 = 0$. By assumption we then have $x_0 > 0$, so (x, y) is a positive multiple of (a, c) and the theorem is trivially true.

Next suppose that $y_0 > 0$, so that $y_j a_j - x_i c_i > 0$ always. We have two cases:

- If y_i and c_i have the same sign, then by the induction hypothesis $t_i = u_i$; to satisfy the induction step we must show that if c_i changes sign, then y_i does as well. If c_i changes sign, then y_i and a_i have opposite signs so $y_i a_i \leqslant 0$. Therefore $x_i c_i < 0$, so x_i and c_i also have opposite signs, so y_i changes sign.
- If y_i and c_i have opposite signs, then by induction hypothesis $t_i = u_i + 1$. We must show that if y_i changes sign, then c_i does as well. If y_i changes sign, then x_i and c_i have the same sign so $x_i c_i \geqslant 0$. Thus $y_i a_i > 0$, so they have the same sign and c_i changes sign.

The case $y_0 < 0$ proceeds analogously. This completes the proof.

In the case of Bernstein-Yang, the update matrices

$$\begin{pmatrix} 1/2 & 1/2 \\ 0 & 1 \end{pmatrix}, \qquad \begin{pmatrix} 1/2 & 0 \\ 0 & 1 \end{pmatrix}, \qquad \begin{pmatrix} 1/2 & -1/2 \\ 1 & 1 \end{pmatrix}$$

are all ratchet matrices, and y is always odd so it's never zero. Thus they satisfy the assumptions of the theorem.

We wish to count the total number of times s that y_i became negative during an execution of Bernstein-Yang, mod 2. It suffices to count t, the total number of times it changed signs, mod 4. At all times, the parity of t is even if y_i is nonnegative, and odd if it is negative; furthermore $s = \lceil t/2 \rceil$. Here changing t by 2 flips the Jacobi symbol. So we don't need a separate variable to track the Jacobi symbol during updates: that variable can be applied directly to t.

During each batch of k steps, y changes signs t_k times, and c changes signs u_k times. We can directly count u_k, but not t_k. Since we know that $t_k - u_k \in \{0, 1\}$, we can update t by adding u_k to it, and then at the end of the batch, by adding 1 if the sign of y doesn't match the parity of t. In principle, the upper bit of u_k and t can be merged, because they will only be added (xor'ed) together.

Note that Bernstein-Yang can be applied recursively for sub-quadratic runtime. Our approach can also be applied recursively to calculate u_k mod 4, with c_i and u_k taking the places of y_i and t_k, respectively.

To use the theorem recursively, we must check that the working variables for (a, c) satisfy the hypotheses on (x, y), meaning that either $c \neq 0$ or $c = 0 < a$. At the beginning, $(a, c) = (1, 0)$, and we note by induction that a is always strictly odder than c. Changes that maintain $c = 0$ divide a by 2, so it remains positive, maintaining $c = 0 < a$. Once the state leaves $c = 0$, it can never return, and so will always satisfy $c \neq 0$. This is because the first two matrices don't modify c, and the last one changes it to $a + c$. We can't have $a + c = 0$ because a is always odder than c.

5 Reference implementation

A reference Python implementation of our technique is given in Figure 1. It uses the half-delta variant [HAS21,WMt] with batch size $k = 32$ and is not recursive.

6 Conclusion

We demonstrated how to modify the Bernstein-Yang inversion algorithm so that it also computes Jacobi symbols. The modification has minimal cost, and in particular does not change the subquadratic asymptotic performance of Bernstein-Yang. For future work, it would be interesting to compare its conecrete performance with existing Jacobi symbol algorithms.

7 Acknowledgments

Thanks to Paul Zimmermann for helpful feedback.

8 Intellectual property disclosure

Some of these techniques may be covered by US and/or international patents.

References

BY19. Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):340–398, May 2019.

BZ10. Richard P. Brent and Paul Zimmermann. An o(m(n) logn) algorithm for the jacobi symbol. In Guillaume Hanrot, François Morain, and Emmanuel Thomé, editors, *Algorithmic Number Theory*, pages 83–95, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

Ham20. Mike Hamburg. Faster Montgomery and double-add ladders for short Weierstrass curves. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(4):189–208, Aug. 2020. `https://ia.cr/2020/437`.

HAS21. Benjamin Salling Hvass, Diego F. Aranha, and Bas Spitters. High-assurance field inversion for curve-based cryptography. Cryptology ePrint Archive, Report 2021/549, 2021. `https://ia.cr/2021/549`.

Mö19. Niels Möller. Efficient computation of the Jacobi symbol, 2019.

Ste67. Josef Stein. Computational problems associated with Racah algebra. *Journal of Computational Physics*, 1(3):397–405, 1967.

WMt. Peter Wuille, Gregory Maxwell, and `roconnor-blockstream`. Bounds on divsteps iterations in safegcd. `https://github.com/sipa/safegcd-bounds`, accessed 22 September 2021.

```python
from math import log, ceil

def batch_matrix(x,y,delta,batch):
    """Compute matrix using (batch+2) bits of (x,y)"""
    (ai,bi,ci,di,u) = (1,0,0,1,0)

    for i in  range(batch):
        yi = y
        if delta >= 0 and x&1:
            (delta,x,y) = (-delta, (x-y)>>1, x)
            (ai,bi,ci,di) = (ai-ci, bi-di, 2*ai, 2*bi)
        elif x&1:
            (delta,x,y) = (1+delta, (x+y)>>1, y)
            (ai,bi,ci,di) = (ai+ci, bi+di, 2*ci, 2*di)
        else:
            (delta,x,y) = (1+delta, x>>1, y)
            (ai,bi,ci,di) = (ai, bi, 2*ci, 2*di)

        u += ((yi & y)^(y>>1)) & 2 # quadratic reciprocity
        u += (u&1) ^  int(ci < 0)  # count sign changes
        u %= 4

    return (u,delta,(ai,bi,ci,di))

def jacobi(x,y,batch=32):
    """Return jacobi symbol(x on y)"""
    assert (y>0) and (y&1) # y-input must be positive and odd
    x,delta,t = x%y,0,0

    # Compute number of iterations per:
    # https://eprint.iacr.org/2021/549.pdf page 14
    # https://github.com/sipa/safegcd-bounds
    nbits =  int(ceil(log(y,2)))
    niters = (45907*nbits+26313)//19929

    mask = (1<<(batch+2))-1 # low bits
    for i in  range(0,niters,batch):
        u,delta,(ai,bi,ci,di) = \
            batch_matrix(x & mask, y & mask, delta, batch)
        (x,y) = ((ai*x + bi*y)>>batch, (ci*x + di*y)>>batch)
        t = (t+u) % 4
        t = ( t + ((t&1) ^  int(y<0)) ) % 4

    t = (t+(t&1)) % 4 # snap to [0,2]
    if y in [-1,1]: jacobi = 1-t
    else:           jacobi = 0 # gcd != 1
    return jacobi
```

Fig. 1. Python reference