Optimized Computation of the Jacobi Symbol

Jonas Lindstrøm & Kostas Kryptos Chalkias

Mysten Labs
{jonas, kostas}@mystenlabs.com

Abstract. The Jacobi Symbol is an essential primitive in cryptographic applications such as primality testing, integer factorization, and various encryption schemes. By exploring the interdependencies among modular reductions within the algorithmic loop, we have developed a refined method that significantly enhances computational efficiency. Our optimized algorithm, implemented in the Rust language, achieves a performance increase of 72% over conventional textbook methods and is twice as fast as the previously fastest known Rust implementation.
This work not only provides a detailed analysis of the optimizations but also includes comprehensive benchmark comparisons to illustrate the practical advantages of our methods. Our algorithm is publicly available under an open-source license, promoting further research on foundational cryptographic optimizations.

1 Introduction

The Jacobi symbol, named after the esteemed German mathematician Carl Gustav Jacob Jacobi (1804-1851), is a generalization of the Legendre symbol, $\left(\frac{a}{m}\right) \in \{-1, 0, 1\}$, where for a prime number m,

$$\left(\frac{a}{m}\right) = \begin{cases} 0 & \text{if } p \mid a \\ 1 & \text{if } a \equiv b^2 \bmod p \text{ for some } b, \\ -1 & \text{if } a \not\equiv b^2 \bmod p \text{ for all } b, \end{cases}$$

The Jacobi symbol extends the Legendre symbol to all odd numbers m using the prime factorisation $m = m_1 \cdots m_n$ of m,

$$\left(\frac{a}{m}\right) = \left(\frac{a}{m_1}\right) \cdots \left(\frac{a}{m_n}\right).$$

Note that for the Jacobi Symbol, if $\left(\frac{a}{m}\right) = -1$ then a is definitely not a square modulo m, because there is at least one m_i such that a is not a square modulo m_i, but if $\left(\frac{a}{m}\right) = 1$ we can only conclude that the number of m_i's such that a is not a square modulo m_i is even, so a may either be a square or not.

The Jacobi symbol has very interesting usages in cryptography, in particular primality testing, where it is used as part of the Solovay-Strassen [11] and Baillie-PSW [1], [9] primality tests and integer factorization [8]. It is also a crucial

element in the widely used Tonelli-Shanks algorithm for computing modular square roots [10] and for algorithms depending on computing modular square roots, e.g. a recent hash function to imaginary class groups, used for verifiable delay functions [4]. There also exist cryptographic schemes where computing the Jacobi symbol is part of the encryption algorithm [5]. Another wide use of the Jacobi symbol is in protocols involving quadratic residues, as a way to non-interactively check whether an integer is a quadratic residue modulo n, which can be critical in protocols like secure voting, lottery systems, identity based and timed-release encryption [3]. In many of the mentioned use cases, the Jacobi symbol is used for rejection sampling and will be called many times, so optimising the computation of it will often have a large impact on performance.

This paper introduces a refined algorithm for the Jacobi symbol, optimized for practical implementation and speed. Our method demonstrates superior performance, significantly outperforming both the traditional textbook algorithm and, to our knowledge, the fastest existing implementation in the Rust programming language.

2 Reference algorithm

There is a widely known recursive algorithm to compute the Jacobi symbol which uses the following three facts about Jacobi symbols:

$$\left(\frac{2a}{m}\right) = \begin{cases} -\left(\frac{a}{m}\right) & \text{if } m \equiv 3, 5 \bmod 8, \\ \left(\frac{a}{m}\right) & \text{if } m \equiv 1, 7 \bmod 8, \end{cases} \tag{1}$$

$$\left(\frac{a}{m}\right) = \left(\frac{b}{m}\right) \quad \text{if } a \equiv b \bmod m, \tag{2}$$

for all $a, m \in \mathbb{Z}$ with m odd. Furthermore, if a is also odd, we have the law of quadratic reciprocity,

$$\left(\frac{a}{m}\right) = \begin{cases} -\left(\frac{m}{a}\right) & \text{if } a \equiv m \equiv 3 \bmod 4, \\ \left(\frac{m}{a}\right) & \text{otherwise.} \end{cases} \tag{3}$$

In each iteration, the trailing zeros may be removed using (1) and reduced modulo m using (2) and the inputs may be swapped using (3) to ensure that the first is larger than the second. On average, the number of iterations is logarithmic to the input sizes, $O(\log a \log m)$. On each iteration, the sign of the Jacobi symbol may change but this can be adjusted using (1) and (3).

The algorithm is presented in Algorithm 1 where it is presented as found in some textbooks, e.g. [7] and [6], and also on Wikipedia [12].

Despite the existence of asymptotically faster algorithms, such as [2], these have not seen widespread practical use, particularly for smaller inputs where their complexity does not translate to real-world efficiency gains.

Algorithm 1 Compute $\left(\frac{a}{m}\right)$ for odd m

```
procedure JacobiBase(a, m)
    a ← a mod m
    t ← 1
    while a ≠ 0 do
        while a is even do
            a ← a ≫ 1
            r ← m mod 8
            if r = 3 or r = 5 then
                t ← −t
            end if
        end while
        Swap(a, m)
        if a mod 4 = m mod 4 = 3 then
            t ← −t
        end if
        a ← a mod m
    end while
    if m ≠ 1 then
        return 0
    end if
    return t
end procedure
```

3 Optimized algorithm

We now present an optimised algorithm, Algorithm 2 which adds four optimisations to Algorithm 1:

1. The computation of r in the inner loop can be moved outside of the loop.
2. The modular reductions modulo 8 and 4 can be avoided completely and replaced with a few binary operations. This is possible because the input is always odd and we are only interested in two questions:
   (a) Is x equivalent to 3 or 5 modulo 8?
   (b) Is x equivalent to 3 modulo 4?

   Since x is always odd, these two questions may be answered from just the second and third bit of the input. For (b), note that $x \equiv 3 \bmod 4$ if and only if the second bit of x is set. A criterion for property (a) is derived in Lemma 1.
3. We may count the number of times the sign of t is changed and only do a single change in the case that this number is odd.
4. Since a and m are swapped after each iteration, some of the modular reductions (in this case just bits) may be reused in the next iteration.

We summarize optimzation 2. in the following lemma:

Lemma 1. *Let x be an odd, positive integer. If we write x in binary form as $x = \sum_{i=0}^{n} x_i 2^i$ for $x_i \in \{0,1\}$. Then*

$$x \equiv 3 \bmod 8 \text{ or } x \equiv 5 \bmod 8 \iff x_1 + x_2 = 1.$$

Proof. First assume that $x \equiv 3 \bmod 8$ or $x \equiv 5 \bmod 8$. Notice that $x \equiv x_0 + 2x_1 + 4x_2 \bmod 8$. Since x is odd, $x_0 = 1$, so since $x_1, x_2 \in \{0,1\}$, $x \equiv 3 \bmod 8$ implies that $x_1 = 1$ and $x_2 = 0$ and $x \equiv 5 \bmod 9$ implies that $x_2 = 1$ and $x_1 = 0$. Since these cases are exclusive, it implies that $x_1 + x_2 = 1$.

Optimization 1. allows r to be computed outside the inner loop, and the following lemma shows that the inner loop is not needed – as stated in optimization 3., we are only interested in the parity of the number of sign changes which is computed in the following lemma:

Lemma 2. *The number of times the sign of t is inverted in Algorithm 1 in one iteration of the loop is*

$$f(a,m) = z(a)((m_1 + m_2) \bmod 2) + a'_1 m_1 \quad (4)$$

$$\equiv z(a)_0(m_1 + m_2) + a'_1 m_1 \mod 2 \quad (5)$$

where x_i is the i'th bit of x counting from 0 and we write $a = 2^{z(a)}a'$ for odd a'.

Proof. Equation 4 follows from Lemma 1 and noticing that the inner loop has exactly $z(a)$ iterations. Equation 5 follows directly from modular arithmetic.

We combine the four optimisations in Algorithm 2. Here, we will use a function $\text{Bit}(x,i)$ which returns the i'th bit (counting from 0) of a non-negative integer x so for example, x is odd if and only if $\text{Bit}(x,0) = 1$. We also use a function $\text{TrailingZeros}(x)$ which returns the number of trailing zeros in the binary representation of an integer x. This function is available in the standard library of many modern programming languages, including Rust.

4 Analysis

The number of iterations of the outer loop of Algorithms 1 and 2 are the same, which is $O(\log a + \log m)$, but the number of operations per iteration differs.

In both cases, TrailingZeros, which can be computed using $\text{TrailingZeros}(a) + 1$ calls to $\text{Bit}(a,\cdot)$[1], must be computed, but all the modular reductions in Algorithm 1 ($\text{TrailingZeros}(a) + 2$ in total) are replaced by at most 3 cheap bit checks. Often, there will only be two calls to Bit in Algorithm 2 because $\text{Bit}(m,2)$ may be omitted in all the cases where z is even, which should be case in about $\frac{2}{3}$'rds of the cases, assuming that the distribution of a is uniform. To see this, notice that the share of odd numbers is $1/2$, the share with exactly two

[1] This is done by calling $\text{Bit}(a,0), \text{Bit}(a,1), \ldots$ until the first time it returns 1.

Algorithm 2 Compute $\left(\frac{a}{m}\right)$ for odd m

procedure JACOBINEW(a, m)
  $a \leftarrow a \bmod m$
  $t \leftarrow 1$
  $m_1 \leftarrow \text{BIT}(m, 1)$
  while $a \neq 0$ **do**
    $z \leftarrow \text{TRAILINGZEROS}(a)$
    $a \leftarrow a \gg z$
    $a_1 \leftarrow \text{BIT}(a, 1)$
    if $(\text{BIT}(z, 0) \wedge (m_1 \oplus \text{BIT}(m, 2))) \oplus (a_1 \wedge m_1)$ **then**
      $t \leftarrow -t$
    end if
    SWAP(a, m)
    $m_1 \leftarrow a_1$
    $a \leftarrow a \bmod m$
  end while
  if $m \neq 1$ **then**
    return 0
  end if
  return t
end procedure

trailing zeros is 1/8, the share with four trailing zeros is 1/32 etc. This gives a geometric series with limit,

$$\frac{1}{2} + \frac{1}{8} + \frac{1}{32} + \cdots = 2/3.$$

Finally, The repeated negations of t in the inner loop are reduced to at most one negation by applying Lemma 2.

5 Implementation

The algorithms detailed in Algorithm 1 and 2 have been developed and tested using the Rust programming language and are publicly available for review[2]. Within the Rust ecosystem, the primary comparable implementation that supports arbitrarily large integers is found in the *num-bigint-dig* library[3], a derivative of the *num-bigint* library[4]. Although the *ark-ff* library[5] implements the Legendre symbol, it does not support the Jacobi symbol yet, and thus was not included in our performance evaluations.

[2] https://github.com/jonas-lj/jacobi-benchmarks
[3] https://github.com/dignifiedquire/num-bigint
[4] https://github.com/rust-num/num-bigint
[5] https://github.com/arkworks-rs/algebra

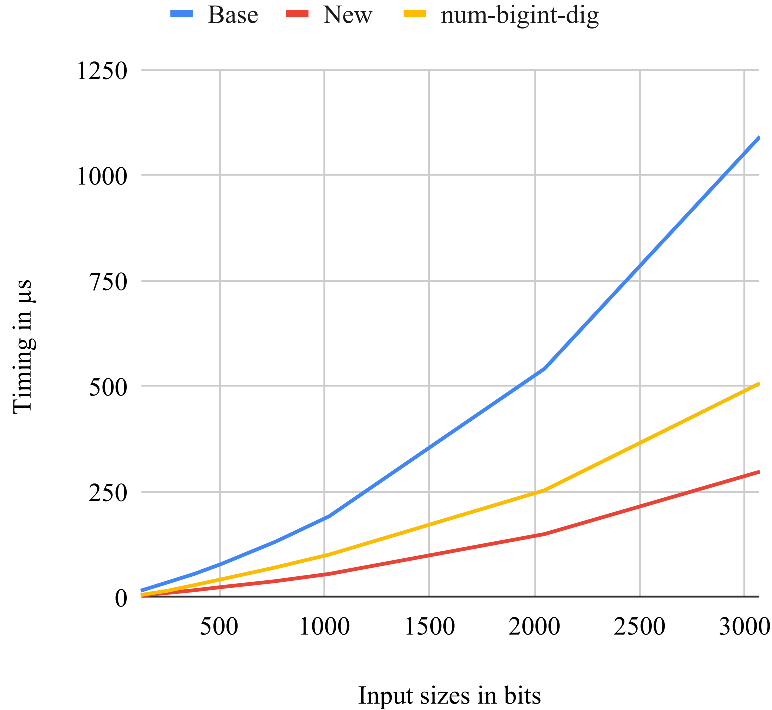

Fig. 1. Benchmarks of Algorithm 1 (Base), Algorithm 2 (New) and *num-bigint-dig*. Both inputs, a and m, have bit length as indicated on the horizontal axis.

The *num-bigint-dig* library offers some optimizations over Algorithm 1, notably optimizations 1. and partly 3. discussed earlier; however, it lacks the additional enhancements we introduce.

Our performance benchmarks were conducted on a range of input sizes frequently used in public key cryptography (128 to 3072 bits), utilizing a MacBook Pro equipped with an Apple M1 Pro CPU. For each modulus, we sample 100 random inputs of the same size (as the modulus) and compute the average time it takes to compute the Jacobi symbol. The results are illustrated in Figure 1.

The results indicate that Algorithm 2 consistently outperforms both the base Algorithm 1 and the *num-bigint-dig* implementation, achieving speed enhancements of 72% and 46% respectively on average across all tested input sizes. Note that since *num-bigint-dig* is a fork, it benefits from access to the internal structure of integers (the limbs) which is not typically permissible with standard dependencies, but is still about 2 times slower than our algorithm. Details on replicating these benchmarks can be found at `https://github.com/jonas-lj/jacobi-benchmarks`.

A Source code of Algorithm 1

The source code provided below is utilized for benchmarking Algorithm 1. It closely mirrors the pseudocode detailed in the references [7] and [6]. This implementation is accessible for review on GitHub. Please note that the displayed code has been streamlined by removing some comments and simplifying certain elements for clarity.

```
pub fn jacobi_base(a: &BigInt, m: &BigInt) -> i8 {
    if !m.is_positive() || m.is_even() {
        panic!("Invalid input");
    }

    let mut a = a.mod_floor(m).into_parts().1;
    let mut m = m.magnitude().clone();

    let mut t = true;

    while !a.is_zero() {
        while a.is_even() {
            a.shr_assign(1);
            let r = m.mod_floor(&BigUint::from(8u8));
            if r == BigUint::from(3u8)
                || r == BigUint::from(5u8) {
                t = !t;
            }
        }
        swap(&mut a, &mut m);
        if a.mod_floor(&BigUint::from(4u8))
             == BigUint::from(3u8)
            && m.mod_floor(&BigUint::from(4u8))
             == BigUint::from(3u8)
        {
            t = !t;
        }
        a.rem_assign(&m);
    }

    if m.is_one() {
        return if t { 1 } else { -1 };
    }
    0
}
```

B Source code of Algorithm 2

The following source code is a Rust implementation of Algorithm 2 and is the source code used in the benchmarks. Note that some comments in the code have been omitted. The implementation uses the *num-bigint* library for integer arithmetic.

```
pub fn jacobi_new(a: &BigInt, m: &BigInt) -> i8 {
    if !m.is_positive() || m.is_even() {
        panic!("Invalid input");
    }

    let mut a = a.mod_floor(m).into_parts().1;
    let mut m = m.magnitude().clone();

    let mut t = true;
    let mut m_1 = m.bit(1);

    while !a.is_zero() {
        let z = a.trailing_zeros().expect("a is not zero");
        if !z.is_zero() {
            a.shr_assign(trailing_zeros);
        }

        let a_1 = a.bit(1);
        if (z.is_odd() && (m_1 ^ m.bit(2))) ^ (m_1 && a_1) {
            t = !t;
        }
        m_1 = a_1;
        swap(&mut a, &mut m);
        a.rem_assign(&m);
    }

    if m.is_one() {
        return if t { 1 } else { -1 };
    }
    0
}
```

C Source code of Jacobi Symbol algorithm from the *num-bigint-dig* library

The following source code is the implemenation of the Jacobi algorithm in the *num-bigint-dig* library as used in the benchmarks. Note that some comments have been removed.

```
pub fn jacobi(x: &BigInt, y: &BigInt) -> isize {
   let mut a = x.clone();
   let mut b = y.clone();
   let mut j = 1;

   if b.is_negative() {
      if a.is_negative() {
         j = -1;
      }
      b = -b;
   }

   loop {
      if b.is_one() {
         return j;
      }
      if a.is_zero() {
         return 0;
      }

      a = a.mod_floor(&b);
      if a.is_zero() {
         return 0;
      }
      let s = a.trailing_zeros().unwrap();
      if s & 1 != 0 {
         let bmod8 = b.get_limb(0) & 7;
         if bmod8 == 3 || bmod8 == 5 {
            j = -j;
         }
      }

      let c = &a >> s;
      if b.get_limb(0) & 3 == 3 && c.get_limb(0) & 3 == 3 {
         j = -j
      }
      a = b;
      b = c;
   }
}
```

References

1. Baillie, R., and Wagstaff, S. S. Lucas pseudoprimes. *Mathematics of Computation 35*, 152 (1980), 1391–1417.
2. Brent, R. P., and Zimmermann, P. An o(m(n) logn) algorithm for the jacobi symbol. In *Algorithmic Number Theory* (Berlin, Heidelberg, 2010), G. Hanrot, F. Morain, and E. Thomé, Eds., Springer Berlin Heidelberg, pp. 83–95.
3. Chalkias, K., Baldimtsi, F., Hristu-Varsakelis, D., and Stephanides, G. Mathematical problems and algorithms for timed-release encryption. *Bulletin of the Transilvania University of Brasov 15.50* (2008), 1–4.
4. Chalkias, K. K., Lindstrøm, J., and Roy, A. An efficient hash function for imaginary class groups. Cryptology ePrint Archive, Paper 2024/295, 2024. https://eprint.iacr.org/2024/295.
5. Cocks, C. C. An identity based encryption scheme based on quadratic residues. In *IMA Conference on Cryptography and Coding* (2001).
6. Cohen, H. *A Course in Computational Algebraic Number Theory*. Springer Publishing Company, Incorporated, 2010.
7. Crandall, R., and Pomerance, C. Prime numbers: A computational perspective. *The Mathematical Gazette 86* (11 2002).
8. Peralta, R., and Okamoto, E. Faster factoring of integers of a special form. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences 79* (1996), 489–493.
9. Pomerance, C., Selfridge, J. L., and Wagstaff, S. S. The pseudoprimes to $25 \cdot 10^9$. *Mathematics of Computation 35*, 151 (1980), 1003–1026.
10. Shanks, D. Five number-theoretic algorithms. pp. 51–70. URL: http://cr.yp.to/bib/entries.html#1973/shanks.
11. Solovay, R., and Strassen, V. A fast monte-carlo test for primality. *SIAM Journal on Computing 6*, 1 (1977), 84–85.
12. Wikipedia. Jacobi symbol — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Jacobi%20symbol&oldid=1203705539, 2024. [Online; accessed 07-March-2024].