Evaluation of Modular Algorithms for High-precision Evaluation of Hypergeometric Constants

Howard Cheng
Department of Mathematics and Computer Science
University of Lethbridge
Lethbridge, Alberta, Canada
Email: howard.cheng@uleth.ca

Christopher Martin
Department of Mathematics and Computer Science
University of Lethbridge
Lethbridge, Alberta, Canada
Email: martinc@uleth.ca

***Abstract*—Many important well-known constants such as π and $\zeta(3)$ can be approximated by a truncated hypergeometric series. A modular algorithm based on rational number reconstruction was previously proposed to reduce space complexity of the well-known binary splitting algorithm [1]. In this paper, we examine some variations of this algorithm using Mersenne number moduli and Montgomery multiplication. Implementations of these variations are compared to existing methods and evaluated for their practicality.**

I. Introduction

We consider the evaluation of the hypergeometric series

$$\sum_{n=0}^{\infty} \frac{a(n)}{b(n)} \prod_{i=0}^{n} \frac{p(i)}{q(i)} \tag{1}$$

to high precision, where a, b, p, and q are polynomials with integer coefficients, and $a(n)$, $b(n)$, $p(n)$, $q(n)$ have bit length $O(\log n)$. In most cases, $b(n) = 1$ and can be omitted and this will be assumed throughout the paper.

These series are commonly used in the high precision evaluation of elementary functions and other constants, including the exponential function, logarithms, trigonometric functions, and constants such as π and the Apéry's constant $\zeta(3)$ [2]. For example, we have the following approximation formulas

$$\frac{1}{\pi} = 12 \sum_{n=0}^{\infty} (-1)^n \frac{545140134n + 13591409}{640320^{3n+3/2}} \frac{(6n)!}{(3n)!n!^3} \tag{2}$$

and

$$\zeta(3) \approx \frac{1}{2} \sum_{n=0}^{N-1} \frac{(-1)^n \left(205\,n^2 + 250\,n + 77\right) \left((n+1)!\right)^5 (n!)^5}{\left((2\,n+2)!\right)^5}. \tag{3}$$

In the latter case, we have $a(n) = 205n^2 + 250n + 77$, $p(0) = 1$, $p(n) = -n^5$ for $n > 0$, and $q(n) = 32(2n+1)^5$.

We also assume that the series is linearly convergent, so that the nth term of (1) is $O(c^{-n})$ with $c > 1$. Thus, we may instead evaluate the truncated hypergeometric series

$$S(N) = \sum_{n=0}^{N-1} \frac{a(n)}{b(n)} \prod_{i=0}^{n} \frac{p(i)}{q(i)}. \tag{4}$$

If d decimal digits of (1) is desired, the number of terms to be computed in (4) is $N = O(d)$.

"Binary splitting" is an approach that has been independently discovered and used by many authors in the computation of (4) [2]–[7]. Binary splitting computes the numerator and denominator of the rational number $S(N)$, and the decimal representation of $S(N)$ is then computed by fixed-point division of the numerator by the denominator. Typically, the numerator and denominator computed by binary splitting have large common factors. For example, in the computation of 640,000 digits of $\zeta(3)$, as much as 86% of the size of the computed numerator and denominator can be attributed to their common factor [8].

It was first shown in [1] that the reduced fraction $S(N)$ has numerator and denominator whose sizes are $O(N)$ instead of the $O(N \log N)$ fraction computed by standard binary splitting. Based on this result, a modular algorithm was given to compute the reduced fraction with the same time complexity as binary splitting but a reduced space complexity. However, the algorithm as stated was not practical. This work was later generalized to show that a wider class of constants including π also has reduced fractions whose sizes are $O(N)$ [9]. The algorithm of [8] was extended to give a practical algorithm with a space complexity of $O(N)$.

In this paper, we examine some variations of the modular algorithm given in [1] in order to make it more practical. In particular, we will examine the use of Mersenne numbers as the chosen modulus, as well as the use of Montgomery multiplication [10], in order to accelerate the modular algorithm. We will show that these techniques make the modular algorithm more practical, especially when the size of the reduced fraction is significantly smaller than that computed by standard binary splitting. However, the algorithm in [9] is still superior.

II. Previous Works

A number of authors have used an approach generally called "binary splitting" to compute the numerator and denominator of (4). The approach recursively computes the sum of the first half and the second half, and combine the results using the

property of the denominators to avoid common denominator computations. Let $P = p(0) \cdot p(n-1)$, $Q = q(0) \cdot q(n-1)$, and $T = S(N) \cdot Q$. Binary splitting computes the triple (P, Q, T) for (4) by dividing the sum into two halves and computing the corresponding triples (P_l, Q_l, T_l) and (P_r, Q_r, T_r) for the left and right halves, respectively. These results are then combined as follows

$$(P, Q, T) = (P_l P_r, Q_l Q_r, Q_r T_l + P_l T_r). \tag{5}$$

To compute d digits ($O(N)$ terms) of (4), binary splitting has a time complexity of $O(M(d \log d) \log d) = O(M(d) \log^2 d)$, where $M(t) = O(t \log t \log \log t)$ is the complexity of multiplication of two t-bit integers [11]. The space complexity is $O(d \log d)$ because the computed numerator and denominator have this size in general. No attempt is made to further reduce the fraction as the terms are combined. We refer the reader to [2] for a more detailed description of this approach and its analysis.

In [8], it was observed experimentally that formula (3) for $\zeta(3)$ computed by binary splitting indeed contains a very large common factor. At the same time, this large common factor is a product of many small primes. To help remove common factors during the binary splitting process, the partially factored representation of integers was introduced. The computed values at the base cases of the recursion were factored by trial division, and the factorizations were preserved as much as possible as the integers are combined by addition and multiplication. A large common factor can be removed simply by examining the partial prime factorization. The use of partially factored representation reduced the run time significantly, although time and space complexities did not improve.

It was later shown that the reduced fraction $S(N)$ has numerator and denominator whose sizes are $O(d)$ instead of $O(d \log d)$ for a large class of hypergeometric series including (2) and (3) [1], [9]. The proofs were based on analyzing the number of times each prime divides into the numerator and denominator, and is in fact related to the partially factored representation introduced in [8]. Two $O(d)$ space algorithms were introduced by [1], [9]. First, a modular algorithm was given in [1] to compute the image of the fraction $S(N) \equiv TQ^{-1} \bmod M$ for an appropriately chosen M with $O(d)$ digits, and the reduced fraction T/Q was recovered using rational number reconstruction [12]–[15]. While the algorithm has the same time complexity as binary splitting, it is only interesting in theory because of the additional overhead in modular computations. Later, the algorithm in [8] using partially factored representation was extended so that the factorization of the values at the base cases where handled efficiently by a sieve rather than by trial division [9]. Although the time complexity of this algorithm is still the same as that of binary splitting, in practice it is significantly faster.

We remark that there are approaches that compute an $O(d)$ size approximation of T and Q using binary splitting, so that $S(N)$ can be approximated to d digits using only $O(d)$ space. However, the results computed by such a process cannot be reused if one wishes to compute more digits by extending the truncated series (i.e. increasing N).

III. Modular Algorithms

In this section, we first describe the modular algorithm given in [1] in more detail. We will also describe two new variations to improve the performance of the modular algorithm using special moduli and Montgomery multiplication.

A. Basic Algorithm

Given positive integers r and m, the rational number reconstruction problem is to find a and b such that $r \equiv ab^{-1} \bmod m$, $\gcd(b, m) = 1$, $|a| < \sqrt{m}/2$, and $0 < b \leq \sqrt{m}$ [12]. The recovered fraction is also reduced. Numerous algorithms exist to solve this problem with a time complexity of $O(M(d) \log d)$ if the bit length of m is $O(d)$ [14], [15].

The modular algorithm is based on rational number reconstruction. It chooses a modulus m such that $2\hat{T}\hat{Q} < m$, where $\hat{T}$ and $\hat{Q}$ are the reduced numerator and denominator. The constraint that $\gcd(Q, m) = 1$ can be enforced by ensuring that smallest prime factor of m exceeds $q_{\max} = \max_{i=0..N-1} |q(i)|$. The integer image $S(N) \bmod m$ is computed, and rational number reconstruction is then applied to recover the reduced fraction $\hat{T}/\hat{Q}$.

The bounds for $\hat{T}$ and $\hat{Q}$ were derived for $\zeta(3)$ in [1] and they have size $O(d)$. Therefore, the modulus m has size $O(d)$. Since m is large, computations modulo m may be slow due to repeated divisions by m. In order to avoid unnecessary divisions, the terms in the summation are grouped into groups of G terms, such that the resulting numerator and denominator computed using the binary splitting approach do not exceed m. The grouping factor G can easily be computed by examining the polynomials $a(n)$, $p(n)$, and $q(n)$, and has $G = O(N/\log N)$ [1]. The algorithm for computing $S(N) \equiv TQ^{-1} \equiv \hat{T}\hat{Q}^{-1} \pmod m$ is given below.

Algorithm 1 Computation of $S(N) \equiv TQ^{-1} \pmod m$

1: Determine the largest grouping factor G such that the values T, P, and Q for the partial sum in the range $[n_1, n_1 + G)$ satisfy $T, P, Q < m$ for any n_1.
2: Divide the range $[0, N)$ into $\lfloor N/G \rfloor$ groups of size G and possibly one additional group of size $N \bmod G$.
3: For each group, compute the values of T, P, and Q using binary splitting.
4: Combine the values computed above modulo m using the same calculations as in binary splitting in (5).
5: Compute $S(N) \equiv TQ^{-1} \pmod m$.

Steps 3 and 4 can be interleaved by using three variables to accumulate the current values of T, P, and Q as we process each group, so we do not need to store the computed values for each group separately.

In the complexity analysis (see [1]), it can be seen that Step 3 has the same time complexity as standard binary splitting although the hidden proportionality constant is smaller. Steps 4 and 5 have a lower complexity as standard binary splitting

but may have high hidden proportionality constants. This is particularly true for Step 5 as it requires a modular inverse computation in addition to multiplication and modular reduction. The same is true for the rational number reconstruction step. Some approaches to reduce the hidden constant in Step 4 are described next.

B. Mersenne Number Modulus

Although Algorithm 1 avoids divisions by m until the results are larger than m, Step 4 of the algorithm still has to combine the values of T, P, and Q from each of the $\lceil N/G \rceil$ groups modulo m. Since m has size $O(d)$, divisions by m can cause a significant overhead as each division requires $O(M(d))$ operations. Since there are approximately $N/G = O(N/(N/\log N)) = O(\log N)$ groups, it follows that Step 4 of the algorithm has a time complexity of $O(M(N)\log N) = O(M(d)\log d)$, and requires both cross multiplications and modular reductions.

One way to reduce the run time of this step is to choose moduli of the form $m = 2^k \pm 1$, so that modular reductions can be performed by bit shifts and additions (note that we cannot use $m = 2^k$ since $\gcd(Q, m) > 1$) [16]. If $0 \le a,\ b < m$, then the modular reduction of $ab \bmod m$ (ignoring the multiplication) can be done in $O(d)$ operations instead of $O(M(d))$ operations. We can write the product $ab = c_1 2^k + c_0$ where $c_0,\ c_1 < 2^k$. Since $2^k \equiv 1 \pmod{2^k - 1}$ and $2^k \equiv -1 \pmod{2^k + 1}$, we can perform the modular reduction $ab \bmod 2^k - 1 = c_1 + c_0 \bmod 2^k - 1$ and $ab \bmod 2^k + 1 = -c_1 + c_0 \bmod 2^k + 1$ in $O(d)$ operations.

We choose to use only Mersenne numbers of the form $m = 2^k - 1$, because the constraint that $\gcd(Q, m) = 1$ can be enforced using the following property [17, Theorem 6.12]:

Theorem 1: If p is an odd prime, then any divisor of the Mersenne number $M_p = 2^p - 1$ is of the form $2kp + 1$ where k is a positive integer.

Thus, if we take p to be a prime such that $p \ge q_{\max}/2$, we can ensure that the smallest prime divisor of the modulus $m = 2^p - 1$ exceeds $q_{\max}$ and hence $\gcd(Q, m) = 1$. In practice, satisfying the constraint that $2\hat{T}\hat{Q} < m$ already implies that $p \ge q_{\max}$, so that we only have to ensure that p is prime.

We also note that we do not need to store the modulus $m = 2^k - 1$ itself, as it is sufficient to know only its bit length k in order to perform computations modulo m. Therefore, using a Mersenne number as the modulus also reduces the space usage in Steps 1–4 of Algorithm 1.

C. Montgomery Multiplication

To further reduce the run time of the modular reductions required in Algorithm 1, we try to replace reductions modulo m by reductions modulo $r = 2^k$ which can be done simply in $O(1)$ operations by truncation of the binary representation using a procedure known as Montgomery multiplication [10].

Suppose that $\gcd(m, r) = 1$ which is satisfied in Algorithm 1, and that $r > m$. Let $m' \equiv -m^{-1} \pmod{r}$, which can be computed once in advance. The following algorithm computes the Montgomery reduction of x, $\text{REDC}(x) = xr^{-1} \bmod m$, provided that $0 \le x \le rm$. Notice that divisions by r can be computed easily with bit operations.

Algorithm 2 Computation of $xr^{-1} \bmod m$

1: $n \leftarrow (x \bmod r)m' \bmod r$
2: $t \leftarrow (x + nm)/r$
3: If $t \le m$ return $t - m$ else return t

To make use of Montgomery multiplication, each integer x is converted into its Montgomery representation $\hat{x} = xr \bmod n = \text{REDC}((x \bmod n)(r^2 \bmod n))$. Additions can be performed directly in this representation. The Montgomery representation of the product $z = xy$ given $\hat{x}$ and $\hat{y}$ can be computed by

$$\hat{z} \equiv (xy)r \equiv ((xr)(yr))r^{-1} \equiv \text{REDC}(\hat{x}\hat{y}) \pmod{n}. \quad (6)$$

To convert $\hat{x}$ back to the standard representation, one can simply use $x = \text{REDC}(\hat{x})$.

Montgomery multiplication is most helpful in situations where the number of conversions between Montgomery and standard representations is small compared to the number of reductions.

IV. Experimental Results

We implemented each of the modular algorithms described above, as well as standard binary splitting. All experiments were done on a dual-core Intel Xeon 2.4 GHz processor with 4GB of RAM, under the Linux operating system. Multiprecision arithmetic was supported by the GMP library version 5.0.2 [18]. Rational reconstruction was done by the Half-GCD algorithm of Lichtblau [15] implemented in GMP.

We first give the results on the computation of π using (2) in Table I. The "Basic Modular" algorithm refers to the algorithm as described in Section III-A without using any special modulus, while "Mersenne + Montgomery" refers to using Montgomery multiplication with m being a Mersenne number. For the computation of π, we see that in fact the modular algorithm is not very competitive compared to standard binary splitting. In fact, the standard binary splitting algorithm can compute more digits than the modular algorithms despite having a higher space complexity. This is due to the fact that the difference between the unreduced fractions computed by binary splitting and the reduced fractions are relatively small II (less than 50%), and the gain in size of the fractions does not overcome to hidden proportionality constants in both the time and space complexity bounds.

The results on the computation of $\zeta(3)$ using (3) are given in Table III. Here the results are reversed—standard binary splitting runs out of memory computing 64 million digits, but the modular algorithms completed the computations. This time, the reduced fractions are significantly smaller (less than 20%) of the unreduced fractions, so that the gain easily outweighs the hidden proportionality constants.

TABLE I
COMPUTATION TIMES FOR π. AN ENTRY OF "?" INDICATES THAT THE COMPUTATION DID NOT COMPLETE DUE TO A LACK OF MEMORY.

Digits	Binary Splitting (s)	Basic Modular (s)	Mersenne (s)	Mersenne + Montgomery (s)
1,000,000	6	30	30	30
2,000,000	15	71	69	72
4,000,000	36	174	168	168
8,000,000	86	427	390	407
16,000,000	201	983	940	976
32,000,000	476	2,278	2,114	2,257
64,000,000	1,034	?	?	?
128,000,000	2,434	?	?	?

TABLE III
COMPUTATION TIMES FOR $\zeta(3)$. AN ENTRY OF "?" INDICATES THAT THE COMPUTATION DID NOT COMPLETE DUE TO A LACK OF MEMORY.

Digits	Binary Splitting (s)	Basic Modular (s)	Mersenne (s)	Mersenne + Montgomery (s)
1,000,000	14	43	39	48
2,000,000	34	102	92	113
4,000,000	83	245	218	271
8,000,000	199	584	507	636
16,000,000	470	1,356	1,194	1530
32,000,000	1138	3,092	2,710	3403
64,000,000	?	6,998	6,159	7054

TABLE IV
BREAKDOWN OF COMPUTATION TIMES FOR $\zeta(3)$ USING THE "BASIC MODULAR" ALGORITHM.

Digits	Steps 1–4 (s)	Step 5 (s)	Rational Reconstruction (s)
1,000,000	16	18	8
2,000,000	39	42	19
4,000,000	94	100	47
8,000,000	231	237	109
16,000,000	542	550	250
32,000,000	1,248	1,252	560
64,000,000	2,870	2,797	1,265

TABLE V
BREAKDOWN OF COMPUTATION TIMES FOR $\zeta(3)$ USING THE "MERSENNE" ALGORITHM.

Digits	Steps 1–4 (s)	Step 5 (s)	Rational Reconstruction (s)
1,000,000	12	18	9
2,000,000	30	42	20
4,000,000	71	98	46
8,000,000	172	228	103
16,000,000	413	531	240
32,000,000	958	1,198	532
64,000,000	2,208	2,710	1,194

TABLE II
SIZE (IN 10^3 BITS) OF COMPUTED RESULTS BY BINARY SPLITTING AND MODULAR ALGORITHM.

	π		$\zeta(3)$	
Digits	T	$\hat{T}$	T	$\hat{T}$
1,000,000	7,511	4,451	31,507	4,218
2,000,000	15,484	8,902	66,347	8,437
4,000,000	31,891	17,805	139,360	16,875
8,000,000	65,629	35,609	292,053	33,752
16,000,000	134,949	71,219	610,772	67,495
32,000,000	277,282	142,438	1,274,876	134,994

For both computations of π and $\zeta(3)$, the cost of Algorithm 1 is dominated by Step 5 and rational reconstruction—more than half the time is spent in these two steps. Breakdowns of the run times for the computation of $\zeta(3)$ by the basic modular algorithm and Mersenne algorithm are shown in Table IV and Table V. As expected, the use of Mersenne numbers as moduli reduces the run time of Steps 1–4 significantly. While the remaining steps are the same in the two variations, we see minor improvements in the last two steps of the "Mersenne" algorithm when d is large. This is due to the reduced memory requirement of Steps 1–4 when Mersenne numbers are used as moduli, so that less work is done in the remaining steps for memory management. Also, Steps 1–4 can be considered as the computation of an approximation of the binary splitting results modulo m. We see that in the basic modular algorithm, this computation is already worse than that of standard binary splitting. By using Mersenne numbers as moduli, this computation is now faster than standard binary

splitting. Of course, the remaining steps introduces additional penalty compared to standard binary splitting, even if the complexities of these steps are better than the binary splitting.

For the computation of $\zeta(3)$ one can see that the penalty for using Mersenne numbers in a modular algorithm instead of binary splitting decreases as the number of digits increase. It is conceivable that for large enough number of digits, the modular algorithm can be superior compared to the basic binary splitting algorithm. However, the same cannot be said about π.

In both cases, the use of Mersenne numbers as moduli is clearly an improvement over the basic modular algorithm. However, the reduction in division times with Montgomery multiplication is overcome by the cost of conversions in the input and output, especially when compared to the modular algorithm using Mersenne numbers as moduli.

These results show that modular algorithms can be made more competitive against standard binary splitting with the use of Mersenne numbers as moduli, especially when the size of the reduced fraction is significantly smaller than that of the unreduced fraction normally computed by binary splitting.

We should also remark that the algorithm given in [9] is generally superior to standard binary splitting in both running time and memory usage as it has much lower proportionality constants in both the time and space complexity estimates than the modular algorithms. As a result, it should still be used in favour of the modular algorithm.

V. Conclusions

In this paper, we examine the modular algorithm for evaluating (4) and studied two variations in an attempt to improve its running time. When the size of the reduced fraction is significantly smaller than that of the unreduced fraction computed by binary splitting, the use of Mersenne numbers in the modular algorithms improves the basic modular algorithm and make it more feasible. Although the modular algorithms are interesting theoretically, it is still advisable to use the linear space algorithm based on binary splitting and factored integer representation given in [9].

References

[1] H. Cheng, B. Gergel, E. Kim, and E. Zima, "Space-efficient evaluation of hypergeometric series," *Communications in Computer Algebra*, vol. 39, no. 2, pp. 41–52, 2005.

[2] B. Haible and T. Papanikolaou, "Fast multiprecision evaluation of series of rational numbers," in *Algorithmic Number Theory, Third International Symposium, ANTS-III*, ser. Lecture Notes in Computer Science, J. P. Buhler, Ed., vol. 1423. Springer, 1998.

[3] J. Borwein and P. Borwein, *Pi and the AGM*. John Wiley and Sons, 1987.

[4] J. M. Borwein, D. M. Bradley, and R. E. Crandall, "Computational strategies for the Riemann zeta function," *Journal of Computational and Applied Mathematics*, vol. 121, pp. 247–296, 2000.

[5] R. P. Brent, "Fast multiple-precision evaluation of elementary functions," *Journal of the ACM*, vol. 23, no. 2, pp. 242–251, 1976.

[6] R. W. Gosper, "Strip mining in the abandoned orefields of nineteenth century mathematics," in *Computers in Mathematics*. Dekker, New York, 1990, pp. 261–284.

[7] E. A. Karatsuba, "Fast evaluation of $\zeta(3)$," *Problemy Peredachi Informatsii*, vol. 27, pp. 68–73, 1993.

[8] H. Cheng and E. V. Zima, "On accelerated methods to evaluate sums of products of rational numbers," in *Proceedings of the 2000 International Symposium on Symbolic and Algebraic Computation*, 2000, pp. 54–61.

[9] H. Cheng, G. Hanrot, E. Thomé, E. Zima, and P. Zimmermann, "Time- and space-efficient evaluation of some hypergeometric constants," in *Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation*, C. W. Brown, Ed., 2007, pp. 85–91.

[10] P. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, pp. 519–521, 1985.

[11] A. Schönhage and V. Strassen, "Schnelle multiplikation großer zahlen," *Computing*, vol. 7, pp. 281–292, 1971.

[12] P. S. Wang, M. J. T. Guy, and J. H. Davenport, "p-adic reconstruction of rational numbers," *SIGSAM Bulletin*, vol. 16, no. 2, pp. 2–3, 1982.

[13] G. E. Collins and M. J. Encarnación, "Efficient rational number reconstruction," *Journal of Symbolic Computation*, vol. 20, no. 3, pp. 287–297, 1995.

[14] X. Wang and V. Y. Pan, "Acceleration of euclidean algorithm and rational number reconstruction," *SIAM Journal on Computing*, vol. 32, no. 2, pp. 548–556, 2003.

[15] D. Lichtblau, "Half-GCD and fast rational recovery," in *Proceedings of the 2005 International Symposium on Symbolic and Algebraic Computation*, 2005, pp. 231–236.

[16] E. Zima and A. Stewart, "Cunningham numbers in modular arithmetic," *Programming and Computer Software*, vol. 33, no. 2, pp. 80–86, 2007.

[17] K. H. Rosen, *Elementary Number Theory and Its Applications*. Addison-Wesley, 1992.

[18] "The GNU multiple precision arithmetic library," http://gmplib.org/.