# Efficient Rational Number Reconstruction*

George E. Collins and Mark J. Encarnación

*Research Institute for Symbolic Computation*
*Johannes Kepler University, A-4040 Linz, Austria*
*{George.Collins, Mark.Encarnacion}@risc.uni-linz.ac.at*

17 September 1994

The paper presents an efficient algorithm for reconstructing a rational number from its residue modulo a given integer. The algorithm is based on a double-digit version of Lehmer's multi-precision extended Euclidean algorithm. While asymptotic complexity remains quadratic in the length of the input, experiments with an implementation show that for small inputs the new algorithm is more than 3 times faster than the algorithm in common use, and is more than 7 times faster for inputs that are 100 words long. A second purpose of the paper is to point out certain errors about rational reconstruction that have appeared in the literature.

## 1 Introduction

The problem of reconstructing a rational number from its residue modulo a given integer can be solved using an algorithm based on the extended Euclidean algorithm. Such an algorithm was presented by Wang for use in computing polynomial partial fraction expansions [15]. The algorithm has since been applied in various contexts including polynomial factorization [2, 16], Gröbner basis computations [11, 14], polynomial interpolation [7], and polynomial gcd computations [4].

In the applications just listed, the inputs to Wang's algorithm are typically multiprecision integers, that is, integers that are too large to be stored in a single computer word, and hence require more complicated structures, such as lists or arrays, for their representation. Multiprecision arithmetic is expensive and one would like to minimize its use. For computing gcds this objective has led to the development of several algorithms, from Lehmer's classic [9] to more recent ones by Jebelean [5, 6], Sorenson [13], and Weber [18]. Although asymptotically slower than Schönhage's algorithm [12], these algorithms are faster for the integers we usually encounter—say those at most 100 words long.

*This work was supported in part by Austrian FWF Grant P8572-PHY.

Since Wang's algorithm is the extended Euclidean algorithm equipped with a different stopping condition, we naturally expect that an efficient extended gcd algorithm can be adapted to reconstruct rational numbers. In this paper we describe how a variant of the algorithm in Jebelean [6] can be so adapted.

In Section 2 we review the problem of rational reconstruction and the solution proposed by Wang, while fixing some notation and terminology along the way. We also discuss certain errors that have appeared in the literature. Section 3 describes a multiprecision Euclidean algorithm for computing gcds that will be the basis of our algorithm. In Section 4 we discuss our algorithm and various details that are essential for an efficient implementation.

## 2 Reconstructing Rational Numbers

Given a rational number $R = A/B$ and a positive integer $M$ relatively prime to $B$, we can easily compute $U \in \mathbb{Z}/(M)$, the integers modulo $M$, such that $R \equiv U \pmod{M}$. What about the reverse direction? That is, given a *modulus* $M$ and a *residue* $U \in \mathbb{Z}/(M)$, can we compute a rational number $R = A/B$ for which the congruence $R \equiv U \pmod{M}$ is satisfied?

What we want is an algorithm that will compute integers $A$ and $B$ satisfying

$$A \equiv B\,U \pmod{M}, \tag{1}$$

and, because of the intended application, also

$$0 \leq |A|,\, B < \sqrt{M/2}, \qquad B \neq 0. \tag{2}$$

If $B$ and $M$ are relatively prime then $B^{-1}$ exists in $\mathbb{Z}/(M)$, and the rational number $R = A/B$ is such that $R \equiv U \pmod{M}$. Wang [15] shows that $A$ and $B$, if they exist, are uniquely determined if we require that they be relatively prime. Wang also observes that a modification of the extended Euclidean algorithm can be used to produce the desired output. The algorithm takes as inputs the integers $M$ and $U$ and returns as output a pair of integers $(A, B)$ satisfying (1) and (2), if such a pair exists. If such a pair does not exist, then the algorithm so indicates by outputting NIL. Figure 1 gives Wang's algorithm, which he calls RATCONVERT in [15].

In his original paper [15], Wang does not prove that his algorithm is correct. Rectifying the situation, Wang *et al.* [17] use number-theoretic arguments to show that if the desired rational exists, then it will be produced by the algorithm. Sasaki and Sasaki [10] give an elementary proof of the same assertion. However, there seems to be some confusion about the output of RATCONVERT when the sought-after rational does *not* exist. Both Wang *et al.* and Sasaki and Sasaki claim that if RATCONVERT returns the pair $(A, B)$—and not NIL—then the desired rational is given by $A/B$. Unfortunately, this is not true. As a counterexample, when RATCONVERT is applied to $M = 12$ and $U = 5$ it will output $(A, B) = (-2, 2)$, but we cannot claim that $-2/2 \equiv 5 \pmod{12}$

Algorithm RATCONVERT

Input: A modulus $M \in \mathbb{Z}$ and a residue $U \in \mathbb{Z}/(M)$.

Outputs: A pair $(A, B)$ of integers, with $B > 0$, such that $A \equiv BU \pmod{M}$ and $|A|, |B| < \sqrt{M/2}$, if such a pair exists. Otherwise, NIL is returned.

$(A_1, A_2) := (M, U)$; $(V_1, V_2) := (0, 1)$;<br>
**while true do**<br>
    **if** $V_2 \geq \sqrt{M/2}$ **then return** NIL;<br>
    **if** $A_2 < \sqrt{M/2}$ **then return** $(\mathrm{sign}(V_2)A_2, |V_2|)$;<br>
    $Q := \lfloor A_1/A_2 \rfloor$; $(A_1, V_1) := (A_1, V_1) - Q(A_2, V_2)$;<br>
    SWAP$(A_1, A_2)$; SWAP$(V_1, V_2)$;<br>
**endwhile**;

Figure 1: Wang's Algorithm

since 2 is not invertible in $\mathbb{Z}/(12)$. Also, $-1 \not\equiv 5 \pmod{12}$. After an exhaustive search, we see that there is no pair $(A, B)$ that satisfies both conditions (1) and (2), and for which $B$ is invertible in $\mathbb{Z}/(12)$. Yet RATCONVERT does not return NIL.

The problem, of course, is that the additional condition

$$\gcd(B, M) = 1, \tag{3}$$

which is equivalent to $B$ being invertible in $\mathbb{Z}/(M)$, was not checked. (By definition, gcds are positive.) This problem was not appreciated in [10, 17], and neither is it mentioned in [7, 11, 14, 15], though the problem is recognized in [16]. A correct rational recovery algorithm therefore comprises two parts: the first computes a pair $(A, B)$ of integers that solves the congruence (1) subject to (2), and the second checks condition (3).

We would also like to point out that the relative primality of $M$ and $U$ is neither sufficient (as suggested in [11, p. 374]) nor necessary (as suggested in [14, p. 131]) for satisfying conditions (1), (2), and (3).

Another shortcoming of the literature is that the following question is not addressed: If the residue $U$ is chosen uniformly at random, what is the probability that there exists a pair $(A, B)$ such that (1), (2), and (3) are satisfied? Knowledge of this probability can be useful in the analysis and design of algorithms that make use of rational reconstruction. It turns out that the probability approaches $6/\pi^2 \approx 0.6079$ as $M \to \infty$, since there are about $M$ pairs $(A, B)$ satisfying (2), of which about $(6/\pi^2)M$ are such that $A$ and $B$ are relatively prime.

(The probability that two random integers are relatively prime approaches $6/\pi^2$; see Knuth [8, p. 324].)

## 3 A Multiprecision Euclidean Algorithm

Even though the numbers to which Euclid's algorithm is applied may be large, the quotients that are computed tend to be small. This suggests that we may be able to compute some of these quotients using only the leading digits of the integers under consideration. Lehmer proposes to compute—using only the leading digits of the integers—a certain sequence of pairs of quotients whose agreement guarantees that the quotients are correct (see Lehmer [9] or Knuth [8]). Collins [1] shows how computing only a single sequence of quotients suffices. To state Collins' result we will need to introduce some notation.

Given two positive integers $A_1$ and $A_2$, with $A_1 \geq A_2$, define the *quotient* and *remainder sequences* of $A_1$ and $A_2$ to be $(Q_1, \ldots, Q_r)$ and $(A_1, \ldots, A_{r+2})$, respectively, where

$$Q_i := \lfloor A_i/A_{i+1} \rfloor, \quad \text{and} \quad A_{i+2} := A_i - Q_i A_{i+1},$$

for $i = 1, \ldots, r$, and $A_{r+2} = 0$. Define the *first* and *second cosequences* of $A_1$ and $A_2$ to be $(U_1, \ldots, U_{r+2})$ and $(V_1, \ldots, V_{r+2})$, respectively, where

$$(U_{i+2}, V_{i+2}) := (U_i, V_i) - Q_i(U_{i+1}, V_{i+1}),$$

for $i = 1, \ldots, r$, with $(U_1, V_1) := (1, 0)$ and $(U_2, V_2) := (0, 1)$.

Now let $a_1$ and $a_2$ be the leading digits of $A_1$ and $A_2$. More precisely, for some $h$ such that $A_2 \geq 2^h$, let

$$a_1 = \lfloor A_1/2^h \rfloor \qquad \text{and} \qquad a_2 = \lfloor A_2/2^h \rfloor. \tag{4}$$

If $(q_1, \ldots, q_s)$ is the quotient sequence of $a_1$ and $a_2$, we want a condition that will tell us when

$$q_i = Q_i. \tag{5}$$

Let $(u_1, \ldots, u_{r+2})$ and $(v_1, \ldots, v_{r+2})$ be the first and second cosequences of $a_1$ and $a_2$, respectively. Collins [1] shows that

$$a_{i+2} \geq |v_{i+2}| \qquad \text{and} \qquad a_{i+1} - a_{i+2} \geq |v_{i+1} - v_{i+2}| \tag{6}$$

implies (5), provided all previous quotients were correct, i.e., provided $q_j = Q_j$, for $j < i$. Note that condition (6) also implies that $u_{i+2} = U_{i+2}$ and $v_{i+2} = V_{i+2}$, so that

$$A_{k+1} = u_{k+1}A_1 + v_{k+1}A_2 \qquad \text{and} \qquad A_{k+2} = u_{k+2}A_1 + v_{k+2}A_2. \tag{7}$$

We have the following variant of the Euclidean algorithm for multiprecision input: From $A_1$ and $A_2$, compute $h$ so that $a_1$ and $a_2$ in (4) are single-precision integers. Compute quotients $q_i$ and cosequence elements $u_{i+2}$ and $v_{i+2}$ as long as condition (6) holds. If $k$ is the largest $i$ for which (6) holds, then we can compute $A_{k+1}$ and $A_{k+2}$ by (7). We can then repeat the process with $A_{k+1}$ and $A_{k+2}$ in place of $A_1$ and $A_2$.

To get a half-extended version of this algorithm, that is, one which also computes $V_i$ such that $A_i \equiv UV_i \pmod{M}$, we set $A_1 = M$, $A_2 = U$, compute

$$V_{k+1} = u_{k+1}V_1 + v_{k+1}V_2 \qquad \text{and} \qquad V_{k+2} = u_{k+2}V_1 + v_{k+2}V_2, \tag{8}$$

and then use $V_{k+1}$ and $V_{k+2}$ in place of $V_1$ and $V_2$ when we repeat the process.

Jebelean [6] observed that restricting $a_1$ and $a_2$ to be single-precision, i.e., to fit in a single computer word, makes $u_{k+2}$ and $v_{k+2}$ each about half a word long. By allowing $a_1$ and $a_2$ to be double-precision, we can expect $u_{k+2}$ and $v_{k+2}$ to be (almost) full single-precision. The advantage of allowing $a_1$ and $a_2$ to be double precision, rather than single precision, is that computing the linear combinations in (7) and (8), which are the most time-consuming operations, will happen about half as often, yet each will require about the same amount of time. The disadvantage is that computing with double-precision integers is, of course, more costly than computing with single-precision integers. However, the advantage outweighs the disadvantage, as we will see later.

Figure 2 gives algorithm DDPCC, which, given the leading words of $A$ and $B$, computes the $u$'s and $v$'s that are needed in (7) and (8). The single-precision integer $\beta$ is the base of our number system. For efficiency, we assume that $\beta$ is a power of 2, i.e., $\beta = 2^\zeta$ for some positive integer $\zeta$. Lines 3–5 of DDPCC compute the sequence elements and lines 6–12 check for termination. DDPCC uses a termination condition that is a simplification of that in Jebelean [6]. Note that lines 3, 4, 5, 7, and 11 should not be implemented in the straightforward way as this would be inefficient; bit shifting and subtraction should be used instead.

## 4 Efficient Rational Number Reconstruction

In this section we discuss an efficient algorithm that, given a modulus $M$ and a residue $U$, computes a pair $(A, B)$ of integers that solves the congruence (1) subject to (2), if such a pair exists. The general idea is to take the multiprecision Euclidean algorithm described in the previous section and modify it for this purpose. Also discussed are methods for checking condition (3)—a check that has been neglected in the literature.

### 4.1 Solving the Congruence

The essential difference between the extended Euclidean algorithm (EEA) and Wang's RATCONVERT is their termination condition. The EEA terminates when the remainder becomes zero, while RATCONVERT terminates when

Algorithm DDPCC

Inputs: Non-negative digits $a_1$, $a_0$, $b_1$, and $b_0$, with $a_1 \geq \beta/2$, $b_1 > 0$, and $a := a_1\beta + a_0 \geq b := b_1\beta + b_0$.

Outputs: Digits $u_1$, $u_2$, $v_1$, and $v_2$. Let $A$ and $B$ be positive integers such that, for some $h \geq 0$, we have $a = \lfloor A/2^h \rfloor$ and $b = \lfloor B/2^h \rfloor$. Then, for some $i$, nearly as large as possible, $A_i = u_1 A + v_1 B$ and $A_{i+1} = u_2 A + v_2 B$, where $A_j$ is the $j$th term of the remainder sequence of $A$ and $B$.

$(u_1, v_1) := (1, 0); \quad (u_2, v_2) := (0, 1);$<br>
**while true do**<br>
    $q := \lfloor (a_1\beta + a_0)/(b_1\beta + b_0) \rfloor$;<br>
    $(c_1\beta + c_0) := (a_1\beta + a_0) - q(b_1\beta + b_0);$<br>
    $(u_3, v_3) := (u_1, v_1) - q(u_2, v_2);$<br>
    **if** $c_1 = 0$ **then**<br>
        **if** $c_0 \geq |v_3|$ **and** $(b_1\beta + b_0) - c_0 \geq |v_2 - v_3|$ **then**<br>
            $(u_1, v_1) := (u_2, v_2); \quad (u_2, v_2) := (u_3, v_3);$<br>
        **return**$(u_1, u_2, v_1, v_2)$;<br>
    **if** $b_1 - c_1 \leq 1$ **then**<br>
        **if** $(b_1\beta + b_0) - (c_1\beta + c_0) < |v_2 - v_3|$ **then**<br>
            **return**$(u_1, u_2, v_1, v_2)$;<br>
    $(a_1, a_0) := (b_1, b_0); \quad (b_1, b_0) := (c_1, c_0);$<br>
    $(u_1, v_1) := (u_2, v_2); \quad (u_2, v_2) := (u_3, v_3);$<br>
**endwhile**;

Figure 2: *D*ouble-*D*igit *P*artial *C*osequence *C*omputation.

ALGORITHM DDRPCC

INPUTS: A positive digit $m$ and non-negative digits $a_1$, $a_0$, $b_1$, and $b_0$, with $a_1 \geq \beta/2$, $b_1 > 0$, and $a := a_1\beta + a_0 \geq b := b_1\beta + b_0$.

OUTPUTS: Digits $u_1$, $u_2$, $v_1$, and $v_2$. Let $A$ and $B$ be positive integers such that, for some $h \geq 0$, we have $a = \lfloor A/2^h \rfloor$ and $b = \lfloor B/2^h \rfloor$. Then, for some $i$, nearly as large as possible, $A_i = u_1 A + v_1 B$ and $A_{i+1} = u_2 A + v_2 B$, where $A_j$ is the $j$th term of the remainder sequence of $A$ and $B$. Furthermore, $|v_2| < m$ or $v_1 \leq 1$.

Figure 3: *D*ouble-*D*igit *R*estricted *P*artial *C*osequence *C*omputation.

either (a) the cosequence element becomes at least as large as the critical value $\sqrt{M/2}$, or (b) the remainder becomes less than this critical value. In effect, DDPCC does several Euclidean steps, so a simple-minded use of DDPCC in RATCONVERT may cause us to "overshoot" the termination condition: it may happen that after forming the linear combinations in (7) and (8), using the $u$'s and $v$'s computed by DDPCC, we find that $V_{k+2} \geq \sqrt{M/2}$, and yet there was a $j < k+2$ such that $A_j < \sqrt{M/2}$ and $V_j < \sqrt{M/2}$. In this case we may wrongly conclude that (1) and (2) cannot be satisfied.

Our solution to this problem is as follows. By (8), since $u_{k+2} \leq v_{k+2}$ and $V_1 \leq V_2$, we have $V_{k+2} \leq 2|v_{k+2}V_2|$. Hence

$$\log_2 |V_{k+2}| \leq \log_2 |v_{k+2}| + \log_2 |V_2| + 1.$$

So we will have $|V_{k+2}| < \sqrt{M/2}$ provided $\log_2 |v_{k+2}| + \log_2 |V_2| + 1 < \log_2 \sqrt{M/2}$. The latter is equivalent to $\log_2 |v_{k+2}| < \log_2 \sqrt{M/2} - \log_2 |V_2| - 1$, and this will hold provided

$$L(v_{k+2}) < \lfloor \log_2 \sqrt{M/2} \rfloor - L(V_2) - 1, \tag{9}$$

where $L(x)$ is the bit length of the integer $x$. (Note that $\lfloor \log_2 \sqrt{M/2} \rfloor = \lfloor \log_2 \lfloor \sqrt{\lfloor M/2 \rfloor} \rfloor \rfloor$.) But $v_{k+2}$ is single-precision ($L(v_{k+2}) \leq \zeta$) so whenever the right-hand side of (9) is greater than $\zeta$, we are in no danger of overshooting and we can use DDPCC without problems.

Now when the size of $V_2$ increases so that the right-hand side of (9) is less than or equal to $\zeta$, we can opt not to use DDPCC anymore, and compute the next remainders and cosequence elements as in RATCONVERT. A better alternative would be to have a version of DDPCC that also ensures that its output $v_{k+2}$ satisfies (9), or more generally that $|v_{k+2}| < m$, for some specified integer $m > 0$. Algorithm DDRPCC, for *D*ouble-*D*igit *R*estricted *P*artial *C*osequence *C*omputation, whose specifications are given in Figure 3, ensures that this additional condition on $v_{k+2}$ holds. For the intended purpose, namely to avoid overshoot, the restriction $|v_{k+2}| < m$ need not prevent DDRPCC from doing one step—hence the additional "... or $v_1 \le 1$" in the output specifications, since $v_1 \le 1$ exactly when DDRPCC has done at most one step, that is, $i = 1$ or $i = 2$.

Corresponding to DDPCC and DDRPCC are DPCC (for *D*igit *P*artial *C*osequence *C*omputation) and DRPCC (for *D*igit *R*estricted *P*artial *C*osequence *C*omputation), which take single digits rather than double digits as inputs. We omit their specifications as they would be almost identical to those of the double-digit procedures. These single-digit procedures are only needed for small integers—those of 4 words or less—as we can use the double-digit procedures as long as the elements of the remainder sequence are at least $2\zeta$ bits long, and this would always be the case for larger inputs.

The main algorithm of this paper, christened ISMC for *I*nteger *S*olution of *M*odular *C*ongruence, is given in Figure 4. For the rest of this section we will discuss the efficient implementation of ISMC and its subalgorithms. We implemented the algorithms using SACLIB (version 1.2, which has not yet been released), a C library of computer algebra algorithms [3], which has a word length of $\zeta = 29$. The multiprecision integers were represented using arrays, rather than lists. All our tests were performed on a DECstation 5000/240 running ULTRIX V4.2A. The source code was compiled using the GNU gcc compiler version 2.5.4 with optimization.

In typical situations, ISMC is applied several times to the same modulus $M$ but to different residues. This why we have $\lfloor\sqrt{M/2}\rfloor$ and $L(\lfloor\sqrt{M/2}\rfloor)$ as inputs, since these values do not have to be recomputed each time we apply ISMC.

The bit lengths of single-precision integers are computed several times during an application of ISMC—such computations take place in lines 5, 7, 10, 15, and 22. We implemented a subroutine called DLOG2 (for *D*igit *LOG*arithm, base *2*) that computes the bit length of a single-precision integer by a binary search of a table of powers of 2, combined with bit-shifting. The average computing time of DLOG2 is about 1.4 $\mu$sec.

The computations in line 10 of ISMC do not require division. All we need to do is put the high-order $2\zeta$ bits of $A_1$ in $a_1$ and $a_0$ (with the higher-order $\zeta$ bits in $a_1$). This can be easily done by shifting, and in a bounded amount of time independent of the length of $A_1$ since the integers are stored in arrays. Similar remarks apply to $A_2$ and to the computations in line 15.

We already mentioned in Section 3 that DDPCC should be implemented using bit-shifting and subtraction. To measure how much faster a shift-and-subtract implementation can be, we generated 100000 sets of inputs to DDPCC where $a_1$, $a_0$, and $b_0$ were 29 bits long, and $b_1$ was 28 bits long. The total time required by a straightforward implementation of DDPCC to process these inputs was 72451 msec, while the shift-and-subtract implementation needed only 4584 msec; this is a speedup ratio of 15.81.

For a comparison of DPCC and DDPCC, we generated 100000 inputs as in the preceding paragraph, and used $a_1$ and $b_1$ as inputs to DPCC as well. DPCC

Algorithm ISMC

Inputs: An integer $M$ and a residue $U \in \mathbb{Z}/(M)$. Also $M' := \lfloor \sqrt{M/2} \rfloor$ and $m := L(M')$. [$L(x)$ is the bit length of $x$.]

Outputs: A pair $(A, B)$ of integers, with $B > 0$, such that $A \equiv BU \pmod{M}$ and $|A|, |B| < \sqrt{M/2}$, if such a pair exists. Otherwise, NIL is returned.

$(A_1, A_2) := (M, U); \quad (V_1, V_2) := (0, 1);$
**while true do**
  **if** $V_2 \geq M'$ **then return** NIL;
  **if** $A_2 < M'$ **then return** $\big(\text{sign}(V_2)A_2, |V_2|\big)$;
  $c := m - L(V_2) - 1;$
  $v_1 := 0;$
  **if** $L(A_1) - L(A_2) < \zeta$ **then**
    **if** $L(A_1) \geq 2\zeta$ **and** $c > \zeta/2$ **then**
      $h := L(A_1) - 2\zeta;$
      $(a_1\beta + a_0) := \lfloor A_1/2^h \rfloor; \quad (b_1\beta + b_0) := \lfloor A_2/2^h \rfloor;$
      **if** $c > \zeta$ **then** $(u_1, u_2, v_1, v_2) :=$ DDPCC$(a_1, a_0, b_1, b_0)$;
      **else** $(u_1, u_2, v_1, v_2) :=$ DDRPCC$(2^{c-1}, a_1, a_0, b_1, b_0)$;
    **else if** $L(A_1) \geq \zeta$ **and** $L(A_1) - L(A_2) < \zeta/2$ **then**
      $h := L(A_1) - \zeta;$
      $a_0 := \lfloor A_1/2^h \rfloor; \quad b_0 := \lfloor A_2/2^h \rfloor;$
      **if** $c > \zeta/2$ **then** $(u_1, u_2, v_1, v_2) :=$ DPCC$(a_0, b_0)$;
      **else** $(u_1, u_2, v_1, v_2) :=$ DRPCC$(2^{c-1}, a_0, b_0)$;
  **if** $v_1 \neq 0$ **then**
    $(A_1, A_2) := (u_1, u_2)A_1 + (v_1, v_2)A_2;$
    $(V_1, V_2) := (u_1, u_2)V_1 + (v_1, v_2)V_2;$
  **else**
    $Q := \lfloor A_1/A_2 \rfloor; \quad (A_1, V_1) := (A_1, V_1) - Q(A_2, V_2);$
    SWAP$(A_1, A_2)$;  SWAP$(V_1, V_2)$;
**endwhile**;

Figure 4: *I*nteger *S*olution of *M*odular *C*ongruence.

| $w$ | $S \times T$ | RATCONVERT | | ISMC1 | | ISMC |
|---|---|---|---|---|---|---|
| 2 | $100 \times 100$ | 5033 | *3.32* | 1783 | *1.18* | 1517 |
| 3 | $100 \times 100$ | 8050 | *3.69* | 2684 | *1.23* | 2183 |
| 4 | $100 \times 100$ | 11434 | *4.18* | 3450 | *1.26* | 2733 |
| 5 | $100 \times 100$ | 15300 | *4.50* | 4516 | *1.33* | 3400 |
| 6 | $100 \times 100$ | 19517 | *4.68* | 5583 | *1.34* | 4167 |
| 7 | $100 \times 100$ | 24150 | *5.01* | 6900 | *1.43* | 4817 |
| 8 | $100 \times 100$ | 29284 | *5.14* | 8183 | *1.44* | 5700 |
| 9 | $100 \times 100$ | 34684 | *5.39* | 9700 | *1.51* | 6433 |
| 10 | $100 \times 100$ | 40516 | *5.48* | 11184 | *1.51* | 7400 |
| 20 | $10 \times 100$ | 12183 | *6.41* | 3317 | *1.75* | 1900 |
| 30 | $10 \times 100$ | 24333 | *6.76* | 6650 | *1.85* | 3600 |
| 40 | $10 \times 100$ | 40550 | *6.97* | 11233 | *1.93* | 5817 |
| 50 | $10 \times 100$ | 60833 | *7.14* | 16800 | *1.97* | 8517 |
| 60 | $10 \times 100$ | 85384 | *7.26* | 23583 | *2.00* | 11767 |
| 70 | $10 \times 100$ | 113950 | *7.35* | 31533 | *2.03* | 15500 |
| 80 | $10 \times 100$ | 146167 | *7.41* | 40633 | *2.06* | 19733 |
| 90 | $10 \times 100$ | 182883 | *7.46* | 50867 | *2.08* | 24500 |
| 100 | $10 \times 100$ | 222800 | *7.48* | 62166 | *2.09* | 29784 |
| 1000 | $10 \times 10$ | 2119400 | *8.01* | 587349 | *2.22* | 264617 |

Table 1: Times for solving the congruence (in milliseconds).

did an average of 7.94 steps, and DDPCC an average of 16.41 steps. The total computing times were 1816 msec for DPCC, and 4584 msec for DDPCC, or an average of 2.29 $\mu$sec per DPCC step, and 2.79 $\mu$sec per DDPCC step. The average length of the output $v_2$ was 11.84 bits for DPCC, and 26.34 bits for DDPCC.

Forming the linear combinations in lines 19 and 20 can be done in one pass through the digits of the multiprecision $A$'s and $V$'s since we know the signs of the results in advance: the $A$'s are all positive while the $V$'s alternate in sign. Instead of forming the products separately and then summing, we form the digit-wise products and sums, and determine the appropriate carries (this is where we need sign information) as we process each digit of the multiprecision integers.

Another thing to be noticed about lines 19 and 20 is that when exactly one partial cosequence computation step is successful—as will happen when we are sufficiently near termination—then $u_1 = 0$, $u_2 = 1$, $v_1 = 1$, and $v_2 = -q$, where $q = \lfloor A_1/A_2 \rfloor$. In this case $A_1$ simply takes the value of $A_2$ and only the new value for $A_2$ needs to be computed (similarly for the $V$'s).

Table 1 shows the performance of our implementations of RATCONVERT, ISMC1, and ISMC applied to several test inputs. ISMC1 is similar to ISMC but uses only DPCC, and not DDPCC, and was implemented to measure the improvement one gets by going from single-digits to double-digits in the partial cosequence computations.

The test inputs were generated as follows. For the values of $w$ given in the first column of Table 1, we generated $S$ random moduli that were $w$ words long, i.e., exactly $\zeta w = 29w$ bits long, and for each modulus we generated $T$ random residues. The values of $S$ and $T$ are given in the second column of the table. The total computing times required by the algorithms to process all $S \times T$ input pairs are displayed in the columns labelled RATCONVERT, ISMC1, and ISMC. The italicized entries are the speedup ratios RATCONVERT/ISMC and ISMC1/ISMC, respectively. For inputs that are 100 words long, we see that ISMC is more than twice as fast as ISMC1, and is almost 7.5 times faster than RATCONVERT. The good speedup ratios even for small inputs is attributable in part to the carefully crafted DLOG2 subroutine.

### 4.2 Checking of the Denominator

The best way to check condition (3)—invertibility of the denominator—will depend on the form of the modulus $M$, of which we will presently discuss two important special cases. However, even when $M$ is not known to be of a particular form, we can still do a little better than computing $\gcd(B, M)$ directly. Let $C = (A - BU)/M$. Then $A = BU + CM$ and $\gcd(B, C) = 1$, from which it follows that

$$\gcd(A, B) = \gcd(B, M). \tag{10}$$

Therefore, we can compute $\gcd(A, B)$ instead of $\gcd(B, M)$, and this will be more efficient since both $A$ and $B$ will be at most about half the length of $M$.

Two particular forms of the modulus $M$ occur frequently—perhaps even exclusively. In applications that use a Hensel lifting technique, the modulus $M$ will be a power of a known prime $p$, and checking condition (3) is particularly easy in this case: simply check whether $p$ divides $B$. In applications that use Chinese remaindering, the modulus $M$ will be a product of distinct known primes, and we now have two possible methods: either (I) compute $\gcd(A, B)$ using the algorithm of Section 3, or (II) for each prime divisor $p$ of $M$, check whether $p$ divides $B$, ignoring any remaining primes as soon as a zero remainder is computed. A theoretical computing time analysis will not reveal which of these two methods is faster, so an experimental comparison is in order. (The implementation of method II whose times are given below exploits the fact that in our applications each $p$ will be at most 15 bits long.)

For certain values of $\ell$ ranging from 5 to 100, we formed the product $M$ of the first $\ell$ 15-bit primes, generated 10000 residues $U$ for each $M$, and applied ISMC to get 10000 pairs $(A, B)$ for each value of $\ell$. (We discarded and replaced those $U$'s for which ISMC returned NIL.) We then measured the time to apply methods I and II to the 10000 pairs of inputs for each value of $\ell$. Table 2 summarizes the results. The value of $\ell$ is given in the first column. The times needed by methods I and II are given in the second and third columns. The

| $\ell$ | I | II | I/II | N |
|---|---|---|---|---|
| 5 | 617 | 300 | *2.06* | 8 |
| 10 | 1583 | 767 | *2.06* | 9 |
| 20 | 3500 | 2583 | *1.36* | 24 |
| 30 | 5900 | 5367 | *1.10* | 32 |
| 40 | 8467 | 9200 | *0.92* | 44 |
| 50 | 11450 | 14334 | *0.80* | 64 |
| 60 | 14650 | 20000 | *0.73* | 80 |
| 70 | 18184 | 27083 | *0.67* | 84 |
| 80 | 22033 | 35467 | *0.62* | 102 |
| 90 | 26133 | 44584 | *0.59* | 107 |
| 100 | 30683 | 54534 | *0.56* | 121 |

Table 2: Times for checking the denominator (in milliseconds).

fourth column gives the ratio of the two times. The column labelled N gives the number of times that $B$ was non-invertible in $\mathbb{Z}/(M)$. Note that the number of times for which the denominator was not invertible was relatively small—less than 1% for $\ell \leq 70$. Given the table, a reasonable conjecture is that the expected value of the numbers in column N is $2(6/\pi^2)\ell$.

The table also suggests that method II should be used for smaller inputs, and method I for larger inputs; in our implementation, both methods took about the same amount of time for $\ell = 35$ primes. However, if we take the time to apply ISMC into account, the speedup we get from using method II for $\ell < 35$ is small. For instance, when $\ell = 10$ the total time needed to apply ISMC was 3216 msec so that the speedup we achieved was $(3216 + 1583)/(3216 + 767) \approx 1.2$. We feel that such a small speedup does not justify the extra programming effort needed to use method II for smaller inputs, so we decided to use method I throughout.

## References


[1] G. E. Collins. Lecture notes on arithmetic algorithms. University of Wisconsin, 1980.

[2] G. E. Collins and M. J. Encarnación. Improved techniques in univariate polynomial factorization. In preparation, 1994.

[3] G. E. Collins *et al.* SACLIB 1.1 User's Guide. Technical Report 93-19, RISC-Linz, Johannes Kepler University, A-4040 Linz, Austria, March 1993.

[4] M. J. Encarnación. On a modular algorithm for computing gcds of polynomials over algebraic number fields. In *Proc. ISSAC '94*, pp. 58–65. ACM Press, 1994.

[5] T. Jebelean. A generalization of the binary gcd algorithm. In *Proc. ISSAC '93*, pp. 111–116. ACM Press, 1993.

[6] T. Jebelean. Improving the multiprecision Euclidean algorithm. In *Proc. DISCO '93*, *LNCS* 722, pp. 45–58. Springer-Verlag, 1993.

[7] E. Kaltofen, Y. N. Lakshman, and J.-M. Wiley. Modular rational sparse multivariate polynomial interpolation. In *Proc. ISSAC '90*, pp. 135–139. ACM Press, 1990.

[8] D. E. Knuth. *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms.* Addison-Wesley, 1981.

[9] D. H. Lehmer. Euclid's algorithm for large numbers. *American Mathematical Monthly*, 45(4):227–233, April 1938.

[10] T. Sasaki and M. Sasaki. On integer-to-rational conversion. *SIGSAM Bulletin*, 26(2):19–21, April 1992.

[11] T. Sasaki and T. Takeshima. A modular method for Gröbner-basis construction over Q and solving system of algebraic equations. *Journal of Information Processing*, 12(4):371–379, 1989.

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

[13] J. Sorenson. Two fast GCD algorithms. *J. Algorithms*, 16(1):110–144, 1994.

[14] C. Traverso. Gröbner trace algorithms. In *Proc. ISSAC '88*, *LNCS* 358, pp. 125–138. Springer-Verlag, 1988.

[15] P. S. Wang. A $p$-adic algorithm for univariate partial fractions. In *Proc. SYMSAC '81*, pp. 212–217. ACM Press, 1981.

[16] P. S. Wang. Early detection of true factors in univariate polynomial factorization. In *Proc. EUROCAL '83*, *LNCS* 162, pp. 225–235. Springer-Verlag, 1983.

[17] P. S. Wang, M. J. T. Guy, and J. H. Davenport. $p$-adic reconstruction of rational numbers. *SIGSAM Bulletin*, 16:2–3, 1982.

[18] K. Weber. The accelerated integer gcd algorithm. Tech. Rep. ICM-9307-55, Kent State University, 1993. To appear in *ACM Trans. Math. Softw.*