J. Symbolic Computation (1995) **20**, 287–297

Efficient Rational Number Reconstruction†

GEORGE E. COLLINS AND MARK J. ENCARNACIÓN‡

Research Institute for Symbolic Computation
Johannes Kepler University, A-4040 Linz, Austria

(Received 20 September 1994)

An efficient algorithm is presented for reconstructing a rational number from its residue modulo a given integer. The algorithm is based on a double-digit version of Lehmer's multiprecision 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.

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 (1981) for use in computing polynomial partial fraction expansions. The algorithm has since been applied in various contexts including polynomial factorization (Wang, 1983; Collins & Encarnación, 1995), Gröbner basis computations (Traverso, 1988; Sasaki & Takeshima, 1989), polynomial interpolation (Kaltofen *et al.*, 1990), and polynomial gcd computations (Encarnación, 1994).

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 (1938) classic to more recent ones by Jebelean (1993*a*, 1993*b*), Sorenson (1994), and Weber (1993). Although asymptotically slower than Schönhage's (1971) algorithm, those algorithms are faster for the integers we usually encounter—say those at most 100 words long.

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 (1993*b*) can be so adapted.

In Section 2 we review the problem of rational reconstruction and the solution proposed

† This research was supported in part by Austrian FWF project no. P8572-PHY.

‡ Present address: Department of Computer Science, University of the Philippines, Quezon City 1101, Philippines. E-mail: mje@engg.upd.edu.ph. E-mail: gcollins, mencarna@risc.uni-linz.ac.at

0747–7171/95/090287+11 $12.00/0

ALGORITHM RATCONVERT

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

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

```
1  (A1, A2) := (M, U);   (V1, V2) := (0, 1);
2  while true do
3      if |V2| ≥ √(M/2) then return NIL;
4      if A2 < √(M/2) then return (sign(V2)A2, |V2|);
5      Q := ⌊A1/A2⌋;   (A1, V1) := (A1, V1) − Q(A2, V2);
6      SWAP(A1, A2);   SWAP(V1, V2);
7  end;
```

Figure 1. Wang's Algorithm

by Wang, while fixing some notation and terminology along the way. 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 $U \in \mathbf{Z}/(M)$, 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* $U \in \mathbf{Z}/(M)$, 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

$$A \equiv BU \pmod{M}, \tag{2.1}$$

and, because of the intended application, also

$$0 \leq |A|, B < \sqrt{M/2}, \qquad B \neq 0. \tag{2.2}$$

If B and M are relatively prime then B^{-1} exists in $\mathbf{Z}/(M)$, and the rational number $R = A/B$ is such that $R \equiv U \pmod{M}$. Wang (1981) 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 (2.1) and (2.2), 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 Wang (1981).

In his original paper, Wang (1981) does not prove that his algorithm is correct. Rectifying the situation, Wang *et al.* (1982) use number-theoretic arguments to show that if the desired rational exists, then it will be produced by the algorithm. Sasaki and Sasaki (1992) 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 & 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 $-2/2 \equiv 5 \pmod{12}$ since 2 is not invertible in $\mathbf{Z}/(12)$. Also, $-1 \not\equiv 5 \pmod{12}$. After an exhaustive search, we see that there is no pair (A, B) that satisfies both conditions (2.1) and (2.2), and for which B is invertible in $\mathbf{Z}/(12)$. Yet RATCONVERT does not return NIL.

The problem, is that the additional condition

$$\gcd(B, M) = 1, \tag{2.3}$$

which is equivalent to B being invertible in $\mathbf{Z}/(M)$, was not checked. (By definition, gcds are positive.) This problem was not discussed in the first two papers on the method (Wang, 1981; Wang *et al.*, 1982), though the problem is mentioned in Wang (1983). However, subsequent papers that use the method (Traverso, 1988; Sasaki & Takeshima, 1989; Kaltofen *et al.*, 1990) do not mention that testing (2.3) is necessary in the absence of other tests. In these papers, the final result (*e.g.*a Gröbner basis) is checked so that erroneous rational recovery would be detected anyway. A correct rational recovery algorithm therefore comprises two parts: the first computes a pair (A, B) of integers that solves the congruence (2.1) subject to (2.2), and the second checks condition (2.3).

We would also like to point out that the relative primality of M and U is neither sufficient (as suggested in Sasaki & Takeshima (1989), p. 374) nor necessary (as suggested in Traverso (1988), p. 131) for satisfying conditions (2.1), (2.2), and (2.3): The example given above with $M = 12$ and $U = 5$ shows that $\gcd(M, U) = 1$ is not sufficient, while the example $M = 12$ and $U = 2$ shows that $\gcd(M, U) = 1$ is not necessary, since the pair $(A, B) = (2, 1)$ satisfies the three conditions in this case.

A natural question that has not been addressed in the literature is the following: If the residue U is chosen uniformly at random, what is the probability that there exists a pair (A, B) such that (2.1), (2.2), and (2.3) are satisfied? The answer is that the probability approaches $6/\pi^2 \approx 0.6079$ as $M \to \infty$. This follows from that fact that there are about M pairs (A, B) satisfying (2.2), of which about $(6/\pi^2)M$ are such that A and B are relatively prime; these are precisely the pairs satisfying the three conditions. (The probability that two random integers are relatively prime is $6/\pi^2$; see Knuth (1981), p. 324.) The proof is completed using the equivalence, in the presence of (2.1), of the relative primality of A and B with (2.3).

Knowledge of this probability can be useful in the analysis and design of algorithms that make use of rational reconstruction. For instance, in the application of the rational reconstruction algorithm described in Encarnación (1994), attempts are made to reconstruct rational numbers from the coefficients of certain polynomials with coefficients in $\mathbf{Z}/(M)$, where M is the product of several primes. If these primes are not sufficient in number, then such rationals are unlikely to exist, and we abort the attempt upon encountering any coefficient for which the three conditions cannot be satisfied. To analyze the algorithm, we would like to know how many coefficients, on the average, we will have to process before we abort the attempt. If we assume that the polynomial has coefficients that are chosen uniformly at random from the elements of $\mathbf{Z}/(M)$, then we can expect to process about $1/(1 - 6/\pi^2) \approx 2.55$ coefficients on the average.

Kaltofen and Rolletschek (1989) discuss a more general congruence relation, in which the bounds on the numerator and denominator need not be the same—in contrast with the uniform bound in (2.2). Although using different bounds may sometimes be more

efficient than using a uniform bound, the improvement would not be significant for our intended applications, so we chose to use a uniform bound for simplicity. In any case, the algorithm we describe in Section 4 can be modified to work with different bounds if this happens to be advantageous for the situation at hand.

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 (1938) or Knuth (1981), pp. 328–330). Collins (1980) shows how computing only a single sequence of quotients suffices. To state Collins' result we will need to introduce some notation.

Given 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

$$a_1 = \lfloor A_1 / 2^h \rfloor \qquad \text{and} \qquad a_2 = \lfloor A_2 / 2^h \rfloor. \tag{3.1}$$

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

$$q_i = Q_i. \tag{3.2}$$

Let $(u_1, \ldots, u_{r+2})$ and $(v_1, \ldots, v_{r+2})$, respectively, be the first and second cosequences of a_1 and a_2. Collins (1980) shows that

$$a_{i+2} \geq |v_{i+2}| \qquad \text{and} \qquad a_{i+1} - a_{i+2} \geq |v_{i+1} - v_{i+2}| \tag{3.3}$$

implies (3.2), provided all previous quotients were correct, *i.e.*, provided $q_j = Q_j$, for $j < i$. Note that condition (3.3) also implies that $u_{i+2} = U_{i+2}$ and $v_{i+2} = V_{i+2}$, so that

$$A_{k+1} = u_{k+1} A_1 + v_{k+1} A_2 \qquad \text{and} \qquad A_{k+2} = u_{k+2} A_1 + v_{k+2} A_2. \tag{3.4}$$

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 (3.1) are single-precision integers. Compute quotients q_i and cosequence elements u_{i+2} and v_{i+2} as long as condition (3.3) holds. If k is the largest i for which (3.3) holds, then we can compute A_{k+1} and A_{k+2} by (3.4). 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 $A_i \equiv U V_i \pmod{M}$, we set $A_1 = M$, $A_2 = U$, compute

$$V_{k+1} = u_{k+1} V_1 + v_{k+1} V_2 \qquad \text{and} \qquad V_{k+2} = u_{k+2} V_1 + v_{k+2} V_2, \tag{3.5}$$

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 k, nearly as large as possible, $A_k = u_1 A + v_1 B$ and $A_{k+1} = u_2 A + v_2 B$, where A_i is the ith term of the remainder sequence of A and B.

```
1  (u1, v1) := (1, 0);   (u2, v2) := (0, 1);
2  while true do
3      q := ⌊(a1β + a0)/(b1β + b0)⌋;
4      (c1β + c0) := (a1β + a0) − q(b1β + b0);
5      (u3, v3) := (u1, v1) − q(u2, v2);
6      if c1 = 0 then
7          if c0 ≥ |v3| and (b1β + b0) − c0 ≥ |v2 − v3| then
8              (u1, v1) := (u2, v2);   (u2, v2) := (u3, v3);
9          return(u1, u2, v1, v2);
10     if b1 − c1 ≤ 1 then
11         if (b1β + b0) − (c1β + c0) < |v2 − v3| then
12             return(u1, u2, v1, v2);
13     (a1, a0) := (b1, b0);   (b1, b0) := (c1, c0);
14     (u1, v1) := (u2, v2);   (u2, v2) := (u3, v3);
15 end;
```

Figure 2. *Double-Digit Partial Cosequence Computation*

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

Jebelean (1993*b*) 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 (3.4) and (3.5), 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 two positive multiprecision integers, computes the u's and v's that are needed in (3.4) and (3.5). The single-precision integer β is the base of our number system. For efficiency, we assume that β is a power of 2, *i.e.*$\beta = 2^\zeta$ for some positive integer ζ. 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 (1993*b*). Note that lines 3, 4, 5, 7, and 11 should not be implemented in a 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 (2.1) subject to (2.2), if such a pair exists. The general idea is to take the multiprecision Euclidean algorithm

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 k, nearly as large as possible, $A_k = u_1 A + v_1 B$ and $A_{k+1} = u_2 A + v_2 B$, where A_i is the ith 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

described in the previous section and modify it for this purpose. Also discussed are methods for checking condition (2.3)—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 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 (3.4) and (3.5), 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 (2.1) and (2.2) cannot be satisfied.

Our solution to this problem is as follows. By (3.5), 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

$$L(v_{k+2}) < \lfloor \log_2 \sqrt{M/2} \rfloor - L(V_2) - 1, \tag{4.1}$$

where $L(x)$ is the bit length of the integer x. (Note that we have $\lfloor \log_2 \sqrt{M/2} \rfloor = \lfloor \log_2 \lfloor \sqrt{\lfloor M/2 \rfloor} \rfloor \rfloor$.) But v_{k+2} is single-precision ($L(v_{k+2}) \leq \zeta$) so whenever the right-hand side of (4.1) is greater than ζ 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 (4.1) is less than or equal to ζ, 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 (4.1), 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 $v_1 \leq 1$" in the output specifications, since $v_1 \leq 1$ exactly when DDRPCC has done at most one step, that is, $i = 1$ or $i = 2$.

ALGORITHM ISMC

INPUTS: A modulus M and a residue $U \in \mathbf{Z}/(M)$. 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 such that $A \equiv BU \pmod{M}$ and $|A|,\ B < \sqrt{M/2}$, with $B > 0$, if such a pair exists. Otherwise, NIL is returned.

```
1  (A1, A2) := (M, U);   (V1, V2) := (0, 1);
2  while true do
3      if |V2| ≥ M' then return NIL;
4      if A2 < M' then return (sign(V2)A2, |V2|);
5      c := m − L(V2) − 1;
6      v1 := 0;
7      if L(A1) − L(A2) < ζ then
8          if L(A1) ≥ 2ζ and c > ζ/2 then
9              h := L(A1) − 2ζ;
10             (a1β + a0) := ⌊A1/2^h⌋;   (b1β + b0) := ⌊A2/2^h⌋;
11             if c > ζ then (u1, u2, v1, v2) := DDPCC(a1, a0, b1, b0);
12             else (u1, u2, v1, v2) := DDRPCC(2^(c−1), a1, a0, b1, b0);
13         else if L(A1) ≥ ζ and L(A1) − L(A2) < ζ/2 then
14             h := L(A1) − ζ;
15             a0 := ⌊A1/2^h⌋;   b0 := ⌊A2/2^h⌋;
16             if c > ζ/2 then (u1, u2, v1, v2) := DPCC(a0, b0);
17             else (u1, u2, v1, v2) := DRPCC(2^(c−1), a0, b0);
18     if v1 ≠ 0 then
19         (A1, A2) := (u1, u2)A1 + (v1, v2)A2;
20         (V1, V2) := (u1, u2)V1 + (v1, v2)V2;
21     else
22         Q := ⌊A1/A2⌋;   (A1, V1) := (A1, V1) − Q(A2, V2);
23         SWAP(A1, A2);   SWAP(V1, V2);
24 end;
```

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

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—since we can use the double-digit procedures as long as the elements of the remainder sequence are at least 2ζ 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. In the next section we will discuss the efficient implementation of ISMC and its subalgorithms.

4.2. IMPLEMENTATION ASPECTS

We implemented the algorithms using SACLIB (version 2.1, which has not yet been released), a C library of algebraic algorithms (Collins *et al.*, 1993), 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 compiler gcc version 2.5.4 with optimization (using the '-O' option).

In typical situations, ISMC is applied several times to the same modulus M but to different residues. This is 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 μsec.

The computations in line 10 of ISMC do not require division. All we need to do is put the high-order 2ζ bits of A_1 in a_1 and a_0 (with the higher-order ζ 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 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 μsec per DPCC step, and 2.79 μsec 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

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

Table 1. Computing times for solving the congruence (in milliseconds)

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.

4.3. Checking the Denominator

The best way to check condition (2.3)—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

$$\gcd(A, B) = \gcd(B, M). \tag{4.2}$$

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 (2.3) 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 reported in this section exploits the fact that in our applications each p will be at most 15 bits long.)

ℓ	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. Computing times for checking the denominator (in milliseconds).

For certain values of ℓ ranging from 5 to 100, we formed the product M of the first ℓ 15-bit primes, generated 10000 residues U for each M, and applied ISMC to get 10000 pairs (A, B) for each value of ℓ, discarding and replacing 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 ℓ. Table 2 summarizes the results. The values of ℓ are given in the first column. The times needed by methods I and II are given in the second and third columns. The fourth column gives the ratio of the two times. The column labelled N gives the number of times that B was non-invertible in $\mathbf{Z}/(M)$. Note that the number of cases for which the denominator was not invertible is 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.

Acknowledgements

We thank the referees for some useful suggestions.

References

Collins, G. E. (1980). *Lecture Notes on Arithmetic Algorithms.* Madison: University of Wisconsin

Collins, G. E. & Encarnación, M. J. (1995). Improved techniques for factoring univariate polynomials. Tech. Rep. 95-19, RISC-Linz, Johannes Kepler University, A-4040 Linz, Austria. Submitted for publication.

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

Encarnación, M. J. (1994). On a modular algorithm for computing gcds of polynomials over algebraic number fields. In *Proceedings of the 1994 International Symposium on Symbolic and Algebraic Computation*, pp. 58–65. ACM Press.

Jebelean, T. (1993*a*). A generalization of the binary gcd algorithm. In *Proceedings of the 1993 International Symposium on Symbolic and Algebraic Computation*, pp. 111–116. ACM Press.

Jebelean, T. (1993*b*). Improving the multiprecision Euclidean algorithm. In *Proceedings of DISCO '93, Lecture Notes in Computer Science* **722**, pp. 45–58. Springer-Verlag.

Kaltofen, E., Lakshman, Y. N., Wiley, J.-M. (1990). Modular rational sparse multivariate polynomial interpolation. In *Proceedings of the 1990 International Symposium on Symbolic and Algebraic Computation*, pp. 135–139. ACM Press.

Kaltofen, E., Rolletschek, H. (1989). Computing greatest common divisors and factorizations in quadratic number fields. *Mathematics of Computation* **53**(188), :697–720.

Knuth, D. E. (1981). *Seminumerical Algorithms: The Art of Computer Programming* **2**. Addison-Wesley.

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

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

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

Schönhage, A. (1971). Schnelle Berechnung von Kettenbruchentwicklungen. *Acta Informatica* **1**:139–144.

Sorenson, J. (1994). Two fast GCD algorithms. *Journal of Algorithms* **16**(1), :110–144.

Traverso, C. (1988). Gröbner trace algorithms. In *Proceedings of the 1988 International Symposium on Symbolic and Algebraic Computation, Lecture Notes in Computer Science* **358**, pp. 125–138. Springer-Verlag.

Wang, P. S. (1981). A *p*-adic algorithm for univariate partial fractions. In *Proceedings of the 1976 Symposium on Symbolic and Algebraic Computation*, pp. 212–217. ACM Press.

Wang, P. S. (1983). Early detection of true factors in univariate polynomial factorization. In *Proceedings of the 1983 European Conference on Computer Algebra, Lecture Notes in Computer Science* **162**, pp. 225–235. Springer-Verlag.

Wang, P. S., Guy, M. J. T., Davenport, J. H. (1982). *p*-adic reconstruction of rational numbers. *SIGSAM Bulletin* **16**:2–3.

Weber, K. (1993). The accelerated integer gcd algorithm. Tech. Rep. ICM-9307-55, Kent State University. To appear in *ACM Transactions on Mathematical Software*.