Division-Free Binary-to-Decimal Conversion

Cyril Bouvier, Paul Zimmermann

HAL Id: hal-00864293

https://hal.inria.fr/hal-00864293v2

Submitted on 21 Jan 2014

Division-Free Binary-to-Decimal Conversion

Cyril Bouvier *, Paul Zimmermann †
* Université de Lorraine, LORIA
† INRIA Nancy

Abstract—This article presents algorithms that convert multiple precision integer or floating-point numbers from radix 2 to radix 10 (or to any radix $b > 2$). Those algorithms, based on the "scaled remainder tree" technique, use multiplications instead of divisions in their critical part. Both quadratic and subquadratic algorithms are detailed, with proofs of correctness. Experimental results show that our implementation of those algorithms outperforms the GMP library by up to 50% (using the same low-level routines).

Index Terms—binary-to-decimal conversion, scaled remainder tree, GMP

---◆---

1 INTRODUCTION

Since computers usually work in binary and humans prefer decimal numbers, the conversion of multiple precision integers from radix 2 to radix 10 — or more generally to any radix $b > 2$ — is a fundamental operation. The binary-to-decimal conversion is implemented in most multiple precision libraries (GMP, PARI/GP, CLN, NTL to cite a few) or computer algebra systems (Maple, Mathematica, Sage). The classical way to perform this conversion is to repeatedly divide the given integer by the output radix b: the remainder gives the least significant digit of the output string, and one continues with the quotient until is becomes zero (see Section 4.4, Method 1a in [1]). A subquadratic version of that algorithm based on long division is well understood too (see Section 1.7.2 of [2]).

An alternate algorithm, which is less known, replaces divisions by multiplications in the critical loop. The idea can be found in [3], [4] and [1, Section 4.4], where Knuth presents an algorithm — Method (2a) — to convert fractional numbers using multiplication, and says "*it is certainly possible to convert between integers and fractions by multiplying or dividing by an appropriate power of b or B*"; however no further details are given. A similar idea is used by Bernstein in [5] to compute $U \bmod p_1$, $U \bmod p_2$, ... where $U, p_1, p_2, ...$ are integers; Bernstein calls the underlying structure a "scaled remainder tree".

In this article we present detailed algorithms based on the "scaled remainder tree" idea, both in the quadratic range (for small numbers) and in the subquadratic range (for large numbers). When given a nonnegative integer a of (at most) k digits in radix b, those algorithms first approximate the radix-b fraction a/b^k by a binary fraction $y/2^n$, then this binary fraction is converted to radix b using a division-free algorithm. In the whole article we assume b fits in a machine word, and is not a power of two, in which case trivial algorithms exist.

E-mail: Cyril.Bouvier@inria.fr, Paul.Zimmermann@inria.fr

The article is organized as follows. After a description in Section 2 of classical algorithms, both in the quadratic and subquadratic case, Section 3 presents the division-free algorithms and their proof of correctness; Section 4 briefly discusses how those algorithms can be used to convert floating-point numbers; finally Section 5 compares our implementation of the algorithms of Sections 3 and 4 to the reference implementation of classical algorithms in GMP [6].

2 CLASSICAL DIVISION-FULL CONVERSION

The literature contains only a few references on multiple precision binary-to-decimal conversion. Apart from [1] and [2], we found an article of Roman Maeder in *The Mathematica Programmer* (volume 6, issue 3, 1996). Most references only describe algorithms with division (except [1], but which does not consider multiple precision division-free conversion). We briefly outline those classical algorithms in this section.

2.1 Quadratic Algorithm

The straightforward quadratic conversion of a multiple precision integer a to radix b just consists of computing $a \bmod b$, which will give the least significant digit in radix b, and repeat with $a' = \lfloor a/b \rfloor$, and so on, until we get zero. If the input a has n words, this algorithm clearly takes $O(n^2)$ time, and the critical loop mainly divides a multi-word integer by one word, since we assumed the radix b fits into one machine word. In GMP, this corresponds to the `mpn_divrem_1` function.

An easy way to speed up this algorithm is to first compute the largest power b^j that fits into a word, divide by b^j in each loop, which will produce a remainder r less than b^j, and convert r into j digits in radix b using an auxiliary routine. This will divide the number of calls to `mpn_divrem_1` by a factor of j, where $j = 9$ on a 32-bit computer, and $j = 19$ on a 64-bit computer for $b = 10$. This method is described in the documentation

of GMP. Note that GMP uses the "scaled remainder tree" technique to convert r into j digits: instead of performing j divisions by b, GMP first approximates the fraction r/b^j, then digits are extracted by multiplications.

2.2 Subquadratic Algorithm

The subquadratic classical conversion is described in detail in [2], and is the algorithm used in GMP 5.1.2. Assuming the input number has at most k digits in radix b, with k a power of two, precompute $b^{k/2}$, $b^{k/4}$, $b^{k/8}$, ... Then compute q and r such that $a = qb^{k/2} + r$ with $0 \leq r < b^{k/2}$, and divide recursively q and r (with remainder) by $b^{k/4}$, and so on until the number of digits to be printed is below a threshold k_t, then we use the naive algorithm described in §2.1. This technique is called "remainder tree" in the literature.

The GMP documentation (in version 5.1.2) mentions that the "scaled remainder tree" technique might be used in the subquadratic case, however *some brief experiments [...] didn't give a noticeable improvement*; this is exactly the idea used in the algorithms we describe below.

3 DIVISION-FREE CONVERSION

3.1 Naive Algorithm

The naive conversion first computes a floating-point approximation $x = y/2^n$ of a/b^k. Since we assumed that a is nonnegative and has at most k digits in radix b, we have $0 \leq a/b^k < 1$, thus the integer part of bx will reveal the most significant digit (in radix b) of a, more precisely the digit of weight b^{k-1}. Then we continue with the fractional part of bx, and so on until we get exactly k digits in radix b. Note: since the output radix b is fixed for a given computation, we do not explicitly state it in the parameters of our functions.

Algorithm 1 Naive conversion

Input: integers a and k such that $a < b^k$
Output: a string s of k digits in radix b (with potential leading zeros)
1: choose an integer n such that $2b^k < 2^n$
2: $y_0 \leftarrow \lfloor (a+1)2^n/b^k \rfloor - 1$
3: return CONVERT_NAIVE(y_0, k, n)
4:
5: **function** CONVERT_NAIVE(y_0, k, n)
6:     **for** $i = 1$ to k **do**
7:         $t_i \leftarrow by_{i-1}$
8:         $s_{k-i} \leftarrow \lfloor t_i/2^n \rfloor$
9:         $y_i \leftarrow t_i \bmod 2^n$
10:     return $(s_{k-1}, \ldots, s_0)$

The main cost in Algorithm 1 is the multiplication of b by y_{i-1} in Step 7; a variant with truncation to reduce this cost is given in Algorithm 2. We prove the correctness of this variant in Theorem 3.

The choice of y_0 in Algorithms 1 and 2 is motivated by the inequality proven in the following lemma.

Algorithm 2 Naive conversion with truncation

Input: integers a and k, such that $k \geq 2$ and $a < b^k$
Output: a string s of k digits in radix b (with potential leading zeros)
1: choose an integer n such that $2kb^k < 2^n$
2: $y_0 \leftarrow \lfloor (a+1)2^n/b^k \rfloor - 1$
3: return CONVERT_TRUNC(y_0, k, n)
4:
5: **function** CONVERT_TRUNC(y_0, k, n)
6:     choose a floating-point value $0 \leq \alpha \leq \log_2(b)$
7:     write n_i for $n - \lfloor i\alpha \rfloor$
8:     **for** $i = 1$ to k **do**
9:         $t_i \leftarrow by_{i-1}$
10:         $s_{k-i} \leftarrow \lfloor t_i/2^{n_{i-1}} \rfloor$
11:         $z_i \leftarrow t_i \bmod 2^{n_{i-1}}$
12:         $y_i \leftarrow \lfloor z_i/2^{n_{i-1}-n_i} \rfloor$
13:     return $(s_{k-1}, \ldots, s_0)$

Lemma 1. *Let a and k be nonnegative integers such that $k \geq 2$ and $a < b^k$. Let n be a nonnegative integer such that $2kb^k < 2^n$. Define y_0 as*

$$y_0 = \lfloor \frac{(a+1)2^n}{b^k} \rfloor - 1. \tag{1}$$

Then, the following inequality stands:

$$a + \frac{1}{2} < \frac{b^k y_0}{2^n} < a + 1.$$

Proof: By definition of y_0, one has

$$y_0 \leq \frac{(a+1)2^n}{b^k} - 1 < y_0 + 1,$$

so one can deduce

$$a + 1 - \frac{2b^k}{2^n} < \frac{b^k y_0}{2^n} \leq a + 1 - \frac{b^k}{2^n}. \tag{2}$$

The condition $2kb^k < 2^n$ with $k \geq 2$ implies $4b^k < 2^n$. It follows $a + 1/2 \leq a + 1 - 2b^k/2^n$. $\qquad \square$

Lemma 2. *Let y_0, k, n be nonnegative integers such that $k \geq 2$, $2kb^k < 2^n$ and*

$$a + \frac{1}{2} < \frac{b^k y_0}{2^n} < a + 1,$$

where a in a nonnegative integer such that $a < b^k$. Then, the output of CONVERT_TRUNC(y_0, k, n) satisfies

$$a = \sum_{i=0}^{k-1} s_i b^i \quad with \ 0 \leq s_i < b, \ \forall i \in [0, k-1].$$

Proof: Using the hypothesis that $b^k y_0 < (a+1)2^n$ and $a < b^k$ and the fact that $n_0 = n$, we have $y_0 < 2^{n_0}$. Moreover, for $i \in [1, k]$, due to line 11, one has $0 \leq z_i < 2^{n_{i-1}}$, so $0 \leq y_i < 2^{n_i}$ after line 12. So, for $i \in [1, k]$, one has $0 \leq y_{i-1} < 2^{n_{i-1}}$, thus $0 \leq s_{k-i} < b$.

For $i \in [1, k]$, the following equality stands:

$$by_{i-1} = s_{k-i}2^{n_{i-1}} + y_i 2^{n_{i-1}-n_i} + r_i,$$

where $r_i := t_i \bmod 2^{n_{i-1}-n_i}$. So,

$$
\begin{aligned}
b^k y_0 &= s_{k-1} 2^n b^{k-1} + b^{k-1} y_1 2^{n-n_1} + b^{k-1} r_1 \\
&= s_{k-1} 2^n b^{k-1} + b^{k-1} r_1 \\
&+ b^{k-2} 2^{n-n_1} (2^{n_1} s_{k-2} + y_2 2^{n_1-n_2} + r_2) \\
&\;\;\vdots \\
&= \sum_{i=1}^{k} s_{k-i} 2^n b^{k-i} + \sum_{i=1}^{k} r_i b^{k-i} 2^{n-n_{i-1}} + y_k 2^{n-n_k}.
\end{aligned}
$$

Using the hypothesis, the following inequalities hold:

$$
a + \frac{1}{2} < \sum_{i=0}^{k-1} s_i b^i + \sum_{i=1}^{k} r_i b^{k-i} 2^{-n_{i-1}} + y_k 2^{-n_k} < a + 1. \quad (3)
$$

As $0 \le y_k < 2^{n_k}$, we have $0 \le y_k 2^{-n_k} < 1$. Moreover, since $0 \le r_i < 2^{n_{i-1}-n_i}$ and $2^\alpha \le b$:

$$
\begin{aligned}
0 &\le \sum_{i=1}^{k} r_i b^{k-i} 2^{-n_{i-1}} < \sum_{i=1}^{k} b^{k-i} 2^{-n_i} = \sum_{i=1}^{k} b^{k-i} 2^{i\alpha-n} \\
&\le k \frac{b^k}{2^n} < \frac{1}{2}.
\end{aligned}
$$

So, one can deduce

$$
a - 1 < \sum_{i=0}^{k-1} s_i b^i < a + 1.
$$

Since both a and $\sum_{i=0}^{k-1} s_i b^i$ are integers, the equality follows. $\square$

Theorem 3. *Algorithm 1 and Algorithm 2 return the k digits of a in base b.*

Proof: The correctness of Algorithm 2 is proven by Lemma 1 and Lemma 2.

Algorithm 1, where no truncation occurs, corresponds exactly to Algorithm 2 with $\alpha = 0$, which yields $n_i = n$ and $r_i = 0$. Then Eq. (3) becomes:

$$
a + \frac{1}{2} < \sum_{i=0}^{k-1} s_i b^i + y_k 2^{-n_k} < a + 1,
$$

which gives $a - 1/2 < \sum_{i=0}^{k-1} s_i b^i < a + 1$ since $0 \le y_k 2^{-n_k} < 1$. In this case we can relax the result of Lemma 1 into $a < b^k y_0 / 2^n < a + 1$, which according to Eq. (2) holds as soon as $2b^k < 2^n$, which is exactly what we choose in Algorithm 1. This proves the correctness of Algorithm 1. $\square$

In Algorithm 2 (with truncation), the condition $2kb^k < 2^n$ can be replaced by $2rb^k < 2^n$, where $r \ge 2$ is an upper-bound on the number of truncations.

Note that the quadratic division-free conversion with truncation is very similar in complexity to the quadratic division-full conversion, since the working size decreases regularly from n to 0 bits throughout the algorithm. The main difference is that in the division-free algorithm, divisions are replaced with multiplications, which are cheaper.

Implementation details

In practice, in Algorithm 2, the digits are computed by blocks of j digits, where j is the largest integer such that b^j fits in a word. This replaces the multiplication by b in line 9 by a multiplication by b^j which has the same cost, as b^j fits in a word, and decreases the number of such multiplications by a factor of j.

The remaining problem is to extract from the s_{k-i}'s the j digits (instead of one digit previously). This is done by computing s_{k-i} separately from the rest of the product. First, notice that s_{k-i} is the highest word of the product of y_{i-1} by b^j and is, up to a carry, the high word of the two-word product of the most significant word of y_{i-1} by b^j. This two-word product can be performed by a series of multiplications by b, b^2, b^3 or b^4 and the digit, or the 2, 3 or 4 digits, can be extracted from the high word with look-up tables. The low word of this two-word product is then added to the rest of the product $b^j y_{i-1}$ (performed with the `mpn_mul_1` function of GMP). If a carry arises, the digits are corrected.

3.2 Subquadratic Algorithm

The idea of the subquadratic algorithm is the following: starting from a n-bit integer y such that $yb^k 2^{-n}$ approximates an integer a of k digits in radix b, one will compute two integers y_h and y_ℓ of about $n/2$ bits each, and corresponding integers k_h, n_h, k_ℓ, n_ℓ, such that $y_h b^{k_h} 2^{-n_h}$ and $y_\ell b^{k_\ell} 2^{-n_\ell}$ approximate respectively the more significant part a_h of a and its less significant part a_ℓ (both in radix b). The following lemma shows that if $yb^k 2^{-n}$ is sufficiently near to a, then $y_h b^{k_h} 2^{-n_h}$ and $y_\ell b^{k_\ell} 2^{-n_\ell}$ — as defined below — are both sufficiently near to a_h and a_ℓ respectively.

Lemma 4. *Let $b \ge 3$ be the output radix. Let $g \ge 2$ be an integer (rationale behind g: bound on the number of recursive calls). Let a, y, k and n be nonnegative integers and $d \le 1$ such that $a < b^k$ and*

$$
a - d < \frac{yb^k}{2^n} < a + 1. \quad (4)
$$

Let k_h, k_ℓ, n_h and n_ℓ be nonnegative integer and define k_0 as $k = k_h + k_0$ and n_0 as $n = n_h + n_0$. The integers k_h, k_ℓ, k_0, n_h, n_ℓ and n_0 should satisfy:

- *$k_0 \ge 2$, $k_\ell \le k$, $n_h \le n$ and $n_\ell \le n$,*
- *$4gb^{k_h} < 2^{n_h}$ and $4gb^{k_\ell} < 2^{n_\ell}$.*

Let us define a_h as $a = a_h b^{k_0} + a_0$ and y_h as $y = y_h 2^{n_0} + y_0$ where $0 \le a_0 < b^{k_0}$ and $0 \le y_0 < 2^{n_0}$. Moreover, let us define a_ℓ as $a_\ell = a \bmod b^{k_\ell}$ (i.e., a_ℓ corresponds to the k_ℓ least significant digits of a in radix b). Finally, let us define y_ℓ as

$$
y_\ell = \lfloor \frac{(b^{k-k_\ell} y) \bmod 2^n}{2^{n-n_\ell}} \rfloor,
$$

which means that after multiplying y by b^{k-k_ℓ}, we extract the n_ℓ bits from position $n - n_\ell$ to $n - 1$.

Then, we have $a_h < b^{k_h}$, $a_\ell < b^{k_\ell}$ and

$$a_h - \frac{1}{4} < \frac{y_h b^{k_h}}{2^{n_h}} < a_h + 1, \tag{5}$$

$$a_\ell - d - \frac{1}{4g} < \frac{y_\ell b^{k_\ell}}{2^{n_\ell}} < a_\ell + 1. \tag{6}$$

Proof: First, $a_\ell < b^{k_\ell}$ is satisfied by definition of a_ℓ, and $a_h < b^{k_h}$ is satisfied by definition of a_h and the fact that $a < b^k$.

Now, we will prove Eq. (5). We have from Eq. (4):

$$\frac{y_h 2^{n_0} + y_0}{2^n} b^k < a_h b^{k_0} + a_0 + 1,$$

thus

$$\frac{y_h b^{k_h}}{2^{n_h}} < a_h + 1,$$

since $a_0 + 1 \le b^{k_0}$. On the other side, again from Eq. (4):

$$\frac{y_h 2^{n_0} + y_0}{2^n} b^k > a_h b^{k_0} + a_0 - d \ge a_h b^{k_0} + a_0 - 1,$$

thus

$$\frac{y_h b^{k_h}}{2^{n_h}} > a_h - \frac{1}{b^{k_0}} - \frac{y_0 b^{k_h}}{2^n}.$$

Since $y_0 < 2^{n_0}$ and $b^{k_0} > 8$ (remember $b \ge 3$ and $k_0 \ge 2$),

$$\frac{1}{b^{k_0}} + \frac{y_0 b^{k_h}}{2^n} < \frac{1}{8} + \frac{b^{k_h}}{2^{n_h}} < \frac{1}{8} + \frac{1}{4g} \le \frac{1}{4},$$

since $g \ge 2$. This proves Eq. (5).

Finally, we will prove Eq. (6). Firstly, we assume $a_\ell - d \ge 0$. We have from Eq. (4):

$$2^n \frac{a - d}{b^{k_\ell}} < y b^{k - k_\ell} < 2^n \frac{a + 1}{b^{k_\ell}}.$$

Writing $a = a_1 b^{k_\ell} + a_\ell$, with a_1 representing the most significant digits, and taking the above inequality modulo 2^n, the terms in a_1 cancel:

$$2^n \frac{a_\ell - d}{b^{k_\ell}} < y b^{k - k_\ell} \bmod 2^n < 2^n \frac{a_\ell + 1}{b^{k_\ell}}.$$

Dividing by $2^{n - n_\ell}$, we get:

$$2^{n_\ell} \frac{a_\ell - d}{b^{k_\ell}} - 1 < y_\ell < 2^{n_\ell} \frac{a_\ell + 1}{b^{k_\ell}},$$

thus:

$$a_\ell - d - \frac{b^{k_\ell}}{2^{n_\ell}} < \frac{y_\ell b^{k_\ell}}{2^{n_\ell}} < a_\ell + 1.$$

We thus prove Eq. (6) for the less significant part. In the case where $a_\ell - d < 0$, the proof above is still valid for:

$$\frac{y_\ell b^{k_\ell}}{2^{n_\ell}} < a_\ell + 1.$$

We also have, since $y_\ell \ge 0$,

$$\frac{y_\ell b^{k_\ell}}{2^{n_\ell}} \ge 0 > a_\ell - d - \frac{1}{4g}.$$

So Eq. (6) holds for every value of a_ℓ. □

Note also that we did not assume that a_h and a_ℓ overlap up to here, i.e., $k_h + k_\ell \ge k$ does not necessarily hold.

Algorithm 3 uses a scaled remainder tree; the main difference with Bernstein's work in [5] is that our scaled remainder tree is asymmetrical, and going to a left subtree requires no multiplication. In this algorithm the parameter a of function CONVERT_REC is only here for proof purpose. This means that a_h and a_ℓ need not be computed at lines 9 and 10 during an actual run of the algorithm.

Algorithm 3 Subquadratic Division-Free Conversion

Input: integer a, radix $b \ge 3$, threshold $k_t \ge 3$
Output: a string s of digits in radix b (with potential leading zeros)
1: **function** CONVERT_REC(a, y, k, n, g)
2:     **if** $k \le k_t$ **then**
3:         **return** CONVERT_TRUNC(y, k, n)
4:     **else**
5:         $k_h \leftarrow \lfloor \frac{k+1}{2} \rfloor$
6:         $k_\ell \leftarrow k - k_h + 1$
7:         Choose $n_h \le n$ such that $4g b^{k_h} < 2^{n_h}$
8:         Choose $n_\ell \le n$ such that $4g b^{k_\ell} < 2^{n_\ell}$
9:         $a_h \leftarrow \lfloor a b^{k_h - k} \rfloor$
10:        $a_\ell \leftarrow a \bmod b^{k_\ell}$
11:        $y_h \leftarrow \lfloor y 2^{n_h - n} \rfloor$
12:        $y_\ell \leftarrow \lfloor \frac{(b^{k - k_\ell} y) \bmod 2^n}{2^{n - n_\ell}} \rfloor$.
13:        $s_h \leftarrow$ CONVERT_REC(a_h, y_h, k_h, n_h, g)
14:        $s_\ell \leftarrow$ CONVERT_REC($a_\ell, y_\ell, k_\ell, n_\ell, g$)
15:        **if** the trailing digit of s_h is $b-1$ and the leading digit of s_ℓ is 0 **then**
16:            $s_h \leftarrow s_h + 1$
17:        **return** $s = \lfloor s_h / b \rfloor b^{k_\ell} + s_\ell$
18:
19: $k \leftarrow \lceil \log_b(a) \rceil$
20: $g \leftarrow \max(\lceil \log_2(k) \rceil + 1, k_t)$
21: Choose n such that $4g b^k < 2^n$
22: $y \leftarrow \lfloor \frac{(a+1) 2^n}{b^k} \rfloor - 1$
23: **return** CONVERT_REC(a, y, k, n, g)

Before we prove the correctness of Algorithm 3, we make a comment: the condition $k_t \ge 3$ ensures that $k \ge 4$ in the recursive calls, thus $k_h \ge 2$ (if $k_h = 1$ then $k_\ell = k$ and the function would loop forever) and that the condition $k_0 = k - k_h \ge 2$ is always true.

Lemma 5. *Assume* CONVERT_REC *is called with nonnegative integers a, y, k, n, g, and a real d such that $d \le 1/2$, $k > k_t$, $g \ge 2$, $a < b^k$ and*

$$a - d < \frac{y b^k}{2^n} < a + 1 \tag{7}$$

then

$$a_h < b^{k_h} \text{ and } a_\ell < b^{k_\ell},$$

$$a_h - \frac{1}{4} < \frac{y_h b^{k_h}}{2^{n_h}} < a_h + 1,$$

$$a_\ell - d - \frac{1}{4g} < \frac{y_\ell b^{k_\ell}}{2^{n_\ell}} < a_\ell + 1. \tag{8}$$

Moreover, for $d \le \frac{1}{4}$ and at most g recursive calls, the inequalities

$$a - \frac{1}{2} < \frac{yb^k}{2^n} < a + 1 \ \text{ and } \ a < b^k \tag{9}$$

hold at any level of the recursion.

Proof: The hypothesis of this Lemma, the choices of k_h and k_ℓ, and lines 7 and 8 of Algorithm 3 allow us to use Lemma 4 to prove Eq. (8).

We can apply the result above recursively to a_h and a_ℓ and so on. So if we start from Eq. (7) with $d \le 1/4$, then after at most g recursive calls, we have at most $t \le g$ successive right nodes, so we have

$$a - \frac{1}{4} - \frac{t}{4g} < \frac{yb^k}{2^n} < a + 1$$

at any level of recursion. This proves Eq. (9). $\qquad\square$

Lemma 6. *Assume* CONVERT_REC *is called with nonnegative integers a, y, k, n, g, and a real d such that $d \le 1/4$, $g \ge 2$, $a < b^k$, $k_t \le g$, $4gb^k < 2^n$, that Eq. (7) is satisfied and that at most g recursive calls are made. Then, the output string s satisfies (identifying s and its value in radix b) $s = a$ or $s = a - 1$ (the case $s = a - 1$ cannot happen if $a = 0$).*

Proof: We consider the tree made of the recursive calls of function CONVERT_REC. Let us call its height h. By hypothesis, we have $h \le g$. Using Lemma 5, we have that Eq. (9) is verified for all nodes of the tree.

We will prove the result by induction on h.

For $h = 0$, we are in the case $k \le k_t$. By hypothesis, we have $k \le k_t \le g$, so $4kb^k < 2^n$. We also have by induction hypothesis $a < b^k$. By using the same proof as for Lemma 2, but with the initial inequality of Eq. (9), we can prove that the output of CONVERT_TRUNC gives $s = a$ or $s = a - 1$. For the case $a = 0$, as the output is always nonnegative, we have necessarily $s = a$.

For $h > 0$, we have by induction hypothesis, $s_h = a_h$ or $s_h = a_h - 1$, and $s_\ell = a_\ell$ or $s_\ell = a_\ell - 1$. But since $k_h + k_\ell = k + 1$, s_h and s_ℓ overlap by one digit. We distinguish the following cases, where we denote by $a_{h,0}$ the trailing digit of a_h, by $s_{h,0}$ the trailing digit of s_h, by $a_{\ell,1}$ the leading digit of a_ℓ (which equals $a_{h,0}$), and by $s_{\ell,1}$ the leading digit of s_ℓ. In the following, we consider values of s_h and s_ℓ just after line 14:

- $s_h = a_h$: no fixup occurs since $s_{\ell,1} = s_{h,0}$ or $s_{\ell,1} = s_{h,0} - 1$, so the case $s_{h,0} = b - 1$ and $s_{\ell,1} = 0$ is impossible ($b \ge 3$). Thus the value s at line 17 equals a or $a - 1$ depending on s_ℓ being a_ℓ or $a_\ell - 1$;
- $s_h = a_h - 1$ and $a_{h,0} > 0$: no fixup occurs since $s_{h,0} = a_{h,0} - 1$, so $s_{h,0} < b - 1$. Thus the value s at line 17 equals a or $a - 1$ depending on s_ℓ being a_ℓ or $a_\ell - 1$;
- $s_h = a_h - 1$ and $a_{h,0} = 0$: in this case $s_{\ell,1} = a_{\ell,1} = 0$ (because if $s_\ell = a_\ell - 1$, $a_\ell > 0$, so the leading digit remains the same) and $s_{h,1} = b - 1$. So a fixup step occurs at line 15. After the fixup, we have $s_h = a_h$ and, as in the first case, the value s at line 17 equals a or $a - 1$ depending on s_ℓ being a_ℓ or $a_\ell - 1$.

So the result is proven by induction on the height of tree made of the recursive calls. $\qquad\square$

Theorem 7. *Assume Algorithm 3 is called with the integer a, then the output string s verifies $s = a$ (identifying s and its value in radix b).*

Proof: Using the inequality of line 21 of Algorithm 3 and Eq. (2), we can show that the call to the function CONVERT_REC at line 23 of Algorithm 3 satisfies Eq. (7) with $d = -3/4$. Moreover, line 20 of Algorithm 3 ensures that at most g recursive calls will be made. So Lemma 6 can be applied, and we have $s = a$ or $s = a - 1$.

We will now show that the case $s = a - 1$ is impossible. Having $s = a - 1$ means that an error of one was done on the least significant digit. The least significant digit is handled by CONVERT_TRUNC (basecase) after (at most) g recursive calls of CONVERT_REC on the low part. This means that when CONVERT_TRUNC is called, we have

$$a - d_0 - \frac{g}{4g} < \frac{yb^k}{2^n} < a + 1,$$

with $d_0 = -3/4$ being the error before the first call to CONVERT_REC, as seen above. Thus, when CONVERT_TRUNC is called, the hypotheses of Lemma 2 are satisfied, thus the output of CONVERT_TRUNC will be a. So no error can be done on the least significant digit, and so we necessarily have $s = a$. $\qquad\square$

Theorem 8. *Given an input a of n bits, Algorithm 3 has complexity $O(M(n) \log n)$, where $M(n)$ is the time needed to multiply two n-bit numbers.*

Proof: Since k_t is a constant, it suffices to consider the recursive case. Since g is a constant too, we have $n_h = n/2 + O(1)$ and similarly for n_ℓ. As mentioned earlier, lines 9 and 10 are not performed in a real implementation. The computation of y_h at line 11 is for free, since it consists of extracting the upper bits of y. The multiplication $b^{k-k_\ell}y$ at line 12 costs $2M(n/2)$ since y has n bits, and b^{k-k_ℓ} has $\sim n/2$ bits. (We can compute b^{k-k_ℓ} on the fly without changing the asymptotic complexity up to a constant multiplicative factor; in practice one will precompute b^{k-k_ℓ}, and ensure the same b^{k-k_ℓ} value is needed at each level of the recursive tree.) Thus if $C(n)$ denotes the cost of the algorithm for n-bit input, we have $C(n) = 2C(n/2) + 2M(n/2) + O(n)$, which yields $C(n) = O(M(n) \log n)$. $\qquad\square$

Using a middle product to compute y_ℓ

The computation of y_ℓ on line 12 of Algorithm 3 can be done by computing a middle product [7]. A product of a $2N$-bit integer by a N-bit integer gives a $3N$-bit result that can be divided in 3 parts of N bits each. A middle product is the computation of the N-bit middle part. The computation of y_ℓ falls into this scheme. By taking $k_h = \lfloor (k+1)/2 \rfloor$ and n_ℓ as small as possible, we ensure $b^{k-k_\ell} < 2^{n-n_\ell+1}$. So, the product on line 12 is the middle product of y (of size n) by b^{k-k_ℓ} (of size $n - n_\ell + 1 \simeq n/2$), as we only need the $n_\ell \simeq n/2$ middle bits.

For small sizes (less than a couple of thousands words) we can use the `mpn_mulmid` function of GMP. For larger sizes we use an FFT middle product. This means that to compute the product of x (of size $n/2$) by y (of size n), one computes $xy \bmod 2^n + 1$ with an FFT algorithm and gets the middle product from the $n/2$ high bits of the n-bit result.

In both cases, by using the middle product, we compute either y_ℓ or $y_\ell - 1$, the latter one with very low probability. So this adds a $-1/(4g)$ term on the leftmost term of Eq. (6) in Lemma 4 and leaves the other terms unchanged. In order to ensure that CONVERT_TRUNC still returns a or $a-1$, we just have to change the condition $4gb^k < 2^n$ into $8gb^k < 2^n$ everywhere in the algorithm.

4 FLOATING-POINT CONVERSION

A floating-point number is represented in a binary computer as $x \cdot 2^e$, where the integer x is the n-bit significand, and e is the exponent. Since the algorithms presented in Section 3 first compute a fractional approximation $y2^{-n}$ of ab^{-k}, when converting a floating-point number we have such an approximation for free, especially when e is near $-n$, which means the floating-point number we consider is in the vicinity of 1 (or -1 if negative).

On the contrary, when using the classical conversion, to obtain the integer a such that ab^{-k} approximates $y2^{-n}$, one will need to multiply y by b^k, and then divide by 2^n. This adds an overhead with respect to the integer conversion routine.

5 EXPERIMENTAL RESULTS

We did implement our algorithms in GMP, using the low-level GMP routines, thus the time differences are due to differences in the algorithms, and not in the low-level arithmetic.

All timings were obtained with GMP 5.1.2 on an AMD Phenom II X2 B55 processor running at 3Ghz, with 8Gb of memory and gcc 4.7.3, with optimization level `-O3`. GMP uses a quadratic algorithm up to GET_STR_PRECOMPUTE_THRESHOLD words (29 words in our experiment). The output radix is always $b = 10$.

5.1 Quadratic Algorithm

Figure 1 compares our implementation of Algorithm 2 with the GMP `mpz_get_str` function, for 1 to 50 words. Our implementation is up to about 55% faster than GMP (between 20 and 28 words), then when the number of words increases the subquadratic algorithm used by GMP wins over Algorithm 2 which has quadratic complexity (the crossover happens around 240 words).

Figure 2 compares the conversion of a floating-point approximation of $2/3$ with GMP and with our implementation of Algorithm 3, for 1 to 100 words. As noted in Section 4, this is the optimal case for the "scaled remainder tree" algorithm, since no initial division is
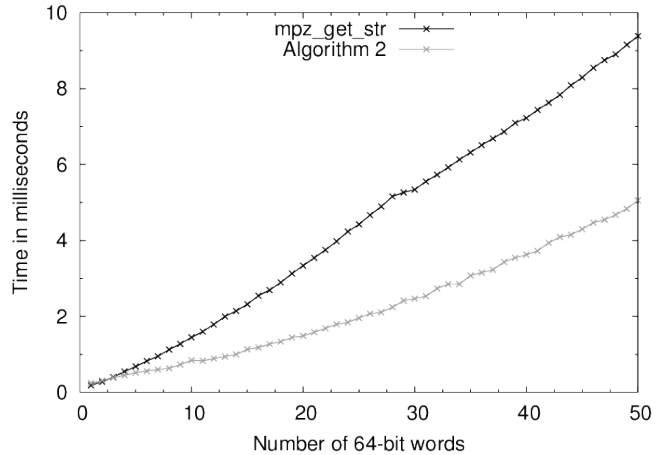

Fig. 1. Integer conversion: comparison with GMP for small sizes.
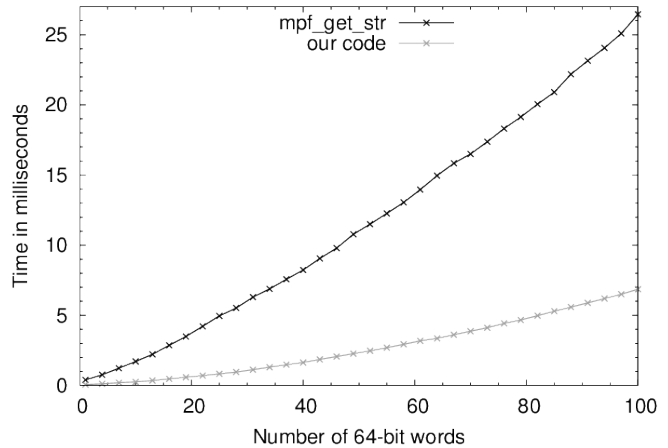

Fig. 2. Floating-point conversion: comparison with GMP for small sizes.

needed, whereas the classical algorithm requires an initial multiplication. The speedup over GMP decreases from 84% (for one word) to 74% (for 100 words). Our algorithm is faster than GMP until around 2100 words.

5.2 Subquadratic Algorithm

For those experiments, the threshold between the quadratic and subquadratic case was $k_t = 3700$, which means that Algorithm 3 outperforms Algorithm 2 up from about 200 words. In Algorithm 3, the GMP `mpn_mulmid` function was used to compute the middle product up to 2500 words, otherwise the middle product was computed with an FFT algorithm.

Figure 3 compares our implementation of Algorithm 3 with the GMP `mpz_get_str` function, up to ten million words. Algorithm 3 outperforms GMP up from about 250000 words. Around ten million words, our implementation is about 19% faster than GMP.

Figure 4 compares the conversion of a floating-point approximation of $2/3$ with GMP and with our implementation of Algorithm 3, up to 50000 words. The
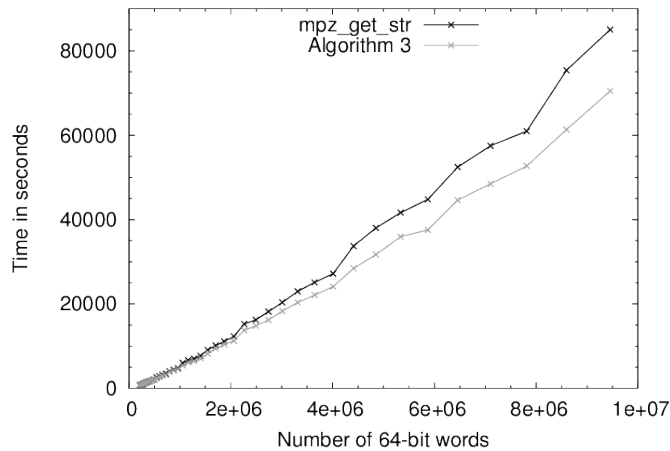
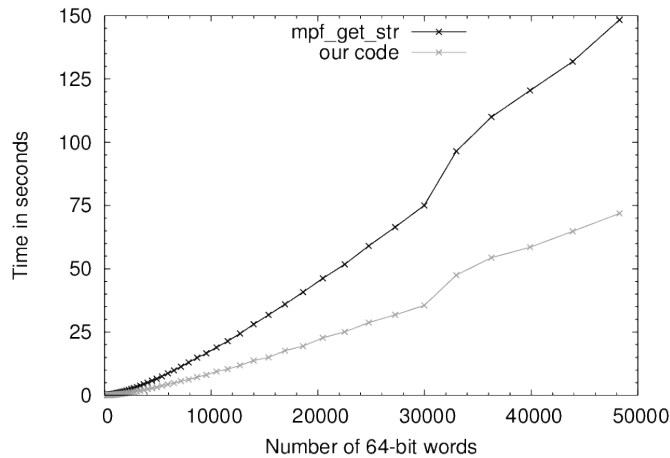Fig. 3. Integer conversion: comparison with GMP for larger sizes.

Fig. 4. Floating-point conversion: comparison with GMP for larger sizes.

speedup over GMP is around 65% for 250 words and then stabilizes around 50% for more than 2500 words. For ten million words, the speedup over GMP is around 55%.

ACKNOWLEDGMENTS

REFERENCES

[1] D. E. Knuth, *The Art of Computer Programming*, 3rd ed. Addison-Wesley, 1998, vol. 2: Seminumerical Algorithms, http://www-cs-staff.stanford.edu/~knuth/taocp.html.

[2] R. P. Brent and P. Zimmermann, *Modern Computer Arithmetic*, ser. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2010, no. 18, electronic version freely available at http://www.loria.fr/~zimmerma/mca/pub226.html.

[3] A. H. Stroud and D. Secrest, "A multiple-precision floating-point interpretive program for the Control Data 1604," *The Computer Journal*, vol. 6, no. 1, pp. 62–66, 1963.

[4] P. H. Sterbenz, *Floating-Point Computation*, ser. in Automatic Computation. Prentice-Hall, 1974.

[5] D. J. Bernstein, "Scaled remainder trees," http://cr.yp.to/papers.html, 2004, 8 pages.

[6] T. Granlund and the GMP development team, *GNU MP: The GNU Multiple Precision Arithmetic Library*, 5th ed., 2013, http://gmplib.org/.

[7] D. Harvey, "The Karatsuba integer middle product," *J. Symb. Comput.*, vol. 47, no. 8, pp. 954–967, 2012.