Division-Free Binary-to-Decimal Conversion

Cyril Bouvier, Paul Zimmermann

Division-Free Binary-to-Decimal Conversion

Cyril Bouvier and Paul Zimmermann

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%.

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 [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 [3] 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.

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 [4].

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

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) says: *The r/b^n scheme described above for using multiplications to bring out digits might be useful for more than a single limb. Some brief experiments with it on the base case when recursing didn't give a noticeable improvement, but perhaps that was only due to the implementation. Something similar would work for the subquadratic divisions too, though there would be the cost of calculating a bigger radix power.* 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 explicit it in the parameters of our functions.

Algorithm 1 Naive conversion

Input: integers a, b, 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 ← ⌊(a+1)2^n / b^k⌋ − 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 ← b y_{i−1}
8:         s_{k−i} ← ⌊t_i / 2^n⌋
9:         y_i ← t_i mod 2^n
10:    return (s_{k−1}, ..., 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 2.

Algorithm 2 Naive conversion with truncation

Input: integers a, b, k, $k \geq 2$, 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 2kb^k < 2^n
2: y_0 ← ⌊(a+1)2^n / b^k⌋ − 1
3: return CONVERT_TRUNC(y_0, k, n)
4:
5: function CONVERT_TRUNC(y_0, k, n)
6:     choose a floating-point value α ≤ log_2(b)
7:     write n_i for n − ⌊iα⌋
8:     for i = 1 to k do
9:         t_i ← b y_{i−1}
10:        s_{k−i} ← ⌊t_i / 2^{n_i − n_{i−1}}⌋
11:        z_i ← t_i mod 2^{n_{i−1}}
12:        y_i ← z_i bdiv 2^{n_{i−1} − n_i}   [truncate the n_{i−1} − n_i least significant bits]
13:    return (s_{k−1}, ..., s_0)
```

Lemma 1: With the notations of Algorithm 2, the following inequalities stand:

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

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

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

The condition $2kb^k < 2^n$ with $k \geq 2$ implies $4b^k < 2^n$. It follows $a + \frac{1}{2} \leq a + 1 - \frac{2b^k}{2^n}$. □

Theorem 2: With the notations of Algorithm 2, $\forall i \in [0, k-1]$, $0 \leq s_i < b$ and

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

Proof: 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 Lemma 1, the following inequalities hold:

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

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 $2^\alpha \le b$:

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

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. □

Algorithm 1, where no truncation occurs, corresponds exactly to Algorithm 2 with $n_i = n$ and $r_i = 0$. Then Eq. (2) 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 condition of Lemma 1 into $a < b^k y_0/2^n < a+1$, which according to Eq. (1) holds as soon as $2b^k < 2^n$, which is exactly what we choose in Algorithm 1. This proves the correctness of Algorithm 1 too.

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

Note: 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 most significant part a_h of a and the least significant part a_ℓ (both in radix b). We first show that if $yb^k 2^{-n}$ is sufficiently near from 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 from a_h and a_ℓ respectively, as defined below.

Remark: In the following, we will write inequalities of the form

$$a - d < \frac{yb^k}{2^n} < a+1, \quad (3)$$

with $d \le 1$. The left-hand side of all these inequalities should be seen modulo b^k, meaning that when $a = 0$, this is equivalent to

$$b^k - d < \frac{yb^k}{2^n} < b^k \text{ or } 0 \le \frac{yb^k}{2^n} < 1.$$

In the following $g \ge 2$ is a fixed integer (rationale behind g: bound on the number of recursive calls). Assume we start from Eq. (3), where a has (at most) k digits in radix b, y has (at most) n bits, $4gb^k < 2^n$ and $d \le 1$.

Choose $k_h, k_\ell < k$, n_h, n_ℓ and define k_0 and n_0 as $k = k_h + k_0$ and $n = n_h + n_0$. The integers k_h, k_ℓ, n_h, and n_ℓ should satisfy:

- $k_0 \ge 2$,
- $4gb^{k_h} < 2^{n_h}$,
- $4gb^{k_\ell} < 2^{n_\ell}$.

Let us write $a = a_h b^{k_0} + a_0$ and $y = y_h 2^{n_0} + y_0$ where $0 \le a_0 < b^{k_0}$ and $0 \le y_0 < 2^{n_0}$. Moreover, let a_ℓ correspond to the k_ℓ least significant digits in radix b of a. Finally, let us define $y_\ell = [(b^{k-k_\ell} y) \bmod 2^n] \operatorname{bdiv} 2^{n-n_\ell}$, which means that after multiplying y by b^{k-k_ℓ}, we extract the n_ℓ bits from position $n-n_\ell$ to $n-1$.

We will first prove that:

$$a_h - \frac{1}{4} < \frac{y_h b^{k_h}}{2^{n_h}} < a_h + 1. \quad (4)$$

We have from Eq. (3):

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

thus

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

since $a_0 + 1 \leq b^{k_0}$. On the other side, again from Eq. (3):

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

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 $b^{k_0} > 8$,

$$\frac{1}{b^{k_0}} + \frac{y_0 b^{k_h}}{2^n} < \frac{1}{8} + \frac{b^{k_h}}{2^{n_h}} < \frac{1}{8} + \frac{1}{4g} \leq \frac{1}{4} \text{ (since } g \geq 2).$$

Thus we get:

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

We will now prove:

$$a_\ell - d - \frac{1}{4g} < \frac{y_\ell b^{k_\ell}}{2^{n_\ell}} < a_\ell + 1. \tag{5}$$

For now, we assume $0 < a_\ell$. We have from Eq. (3):

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

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 (since a_ℓ is an integer, $a_\ell > 0$ and $d \leq 1$, we have $a_\ell - d \geq 0$):

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

Truncating the low $n - n_\ell$ bits, 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 get for the least significant part:

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

In the case where $a_\ell = 0$, the above inequality has to be interpreted modulo b^{k_ℓ}, i.e.:

$$0 \leq \frac{y_\ell b^{k_\ell}}{2^{n_\ell}} < 1 \qquad \text{or} \qquad b^{k_\ell} - d - \frac{1}{4g} < \frac{y_\ell b^{k_\ell}}{2^{n_\ell}} < b^{k_\ell}.$$

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

Algorithm 3 uses a scaled remainder tree [3]; the main difference with Bernstein's work in [3] 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 $b \geq 3$, threshold $k_t \geq 3$
Output: a string s of digits in radix b (with potential leading zeros)

1: **function** CONVERT_REC(a, k, y, n, g)
2: **if** $k \leq k_t$ **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 such that $4gb^{k_h} < 2^{n_h}$
8: Choose n_ℓ such that $4gb^{k_\ell} < 2^{n_\ell}$
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: $y_\ell = [(b^{k-k_\ell} y) \bmod 2^n] \text{ bdiv } 2^{n-n_\ell}$.
13: $s_h \leftarrow$ CONVERT_REC(a_h, k_h, y_h, n_h, g)
14: $s_\ell \leftarrow$ CONVERT_REC($a_\ell, k_\ell, y_\ell, n_\ell, g$)
15: **if** the trailing digit of s_h is $b-1$ and the leading digit of s_ℓ is 0 **then**
16: $s_h \leftarrow s_h + 1 \bmod b^{k_h}$
17: **if** the trailing digit of s_h is 0 and the leading digit of s_ℓ is $b-1$ **then**
18: $s_\ell \leftarrow 000\ldots000$ (k_ℓ times)
19: **return** $s = \lfloor s_h / b \rfloor b^{k_\ell} + s_\ell$
20:
21: $k \leftarrow \lceil \log_b(a) \rceil$
22: $g \leftarrow \max(\lceil \log_2(k) \rceil + 1, k_t)$
23: Choose n such that $4gb^k < 2^n$
24: $y \leftarrow \lfloor \frac{(a+1)2^n}{b^k} \rfloor - 1$
25: return CONVERT_REC(a, k, y, n, g)

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

Lemma 3: Assume CONVERT_REC is called with a, $k > k_t$, y, n, $g \geq 2$. If

$$4gb^k < 2^n \text{ and } a - d < \frac{yb^k}{2^n} < a + 1 \text{ with } d \leq \frac{1}{2}, \tag{6}$$

then

$$4gb^{k_h} < 2^{n_h} \text{ and } 4gb^{k_\ell} < 2^{n_\ell}, \tag{7}$$

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

$$a_\ell - d - \frac{1}{4g} < \frac{yb^{k_\ell}}{2^{n_\ell}} < a + 1.$$

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

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

holds at any level of the recursion.

Proof: Condition (7) is due to lines 7 and 8 of Algorithm 3. The others two conditions are proven in the beginning of this Section.

Now if we start from Eq. (6) with $d \leq 1/4$, then for a left node — i.e., corresponding to some a_h — Eq. (4) holds, and for a right node at the end of a branch of $t \leq g$ successive right nodes, we have

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

□

Lemma 4: Assume CONVERT_REC is called with a, k, y, n, g. Moreover, assume that the input values satisfy condition (6), that $d \leq \frac{1}{4}$ 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 $s = a - 1 \bmod b^k$.

Proof: We consider the tree made of the recursive calls of function CONVERT_REC. Let us call its height h. By hypothesis, we have $h \leq g$. Using Lemma 3, we have, for all nodes of the tree, $a - 1/2 < \frac{yb^k}{2^n} < a + 1$.

We will prove that $s = a$ or $s = a - 1 \bmod b^k$ by induction on h. For $h = 0$, we are in the case $k \leq k_t$ and it follows from the proof of Algorithm 2: indeed if $a + 1/2 < yb^k/2^n < a + 1$, we have seen that necessarily function CONVERT_TRUNC gives $s = a$; and if $a - 1/2 < yb^k/2^n \leq a + 1/2$, it can give $s = a$ or $s = a - 1$ (all others requirements of Theorem 2 are verified). For $h > 0$, we have by induction hypothesis, $s_h = a_h$ or $s_h = a_h - 1 \bmod b^{k_h}$, and $s_\ell = a_\ell$ or $s_\ell = a_\ell - 1 \bmod b^{k_\ell}$. 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. We first note that only one fixup can occur at a time, even when the first one occurs, since in that case s_ℓ stays 0. In the following, except mentioned otherwise, we consider values of s_h and s_ℓ after the recursive calls to CONVERT_REC:

- $s_h = a_h$ and $s_\ell = a_\ell$: no fixup occurs since $s_{h,0} = s_{\ell,1}$, thus the value s at line 19 equals a;
- $s_h = a_h$ and $s_\ell = a_\ell - 1 \bmod b^{k_\ell}$: if $a_\ell = 0$, then $s_{h,0} = a_{h,0} = 0$ and $s_{\ell,1} = b - 1$ and the second fixup occurs, thus $s = a$ at line 19.
  If $a_\ell > 0$, then $s_{h,0} = a_{h,0}$ and $s_{\ell,1} = a_{\ell,1}$. As $a_{h,0} = a_{\ell,1}$, we have $s_{h,0} = s_{\ell,1}$, and no fixup occurs, thus $s = a - 1$ at line 19.
- $s_h = a_h - 1 \bmod b^{k_h}$ and $s_\ell = a_\ell$: in that case $s_{h,0} = s_{\ell,1} - 1 \bmod b$, and the second fixup cannot occur since $b \geq 3$. If $a_{h,0} \neq 0$, then $s_{h,0} \neq b - 1$, thus the first fixup cannot occur, in that case $s = a$ at line 19. Now if $a_{h,0} = 0$, the first fixup occurs, and we get $s = a$ at line 19;
- $s_h = a_h - 1 \bmod b^{k_h}$ and $s_\ell = a_\ell - 1 \bmod b^{k_\ell}$: as in the above case, the second fixup cannot occur since $b \geq 3$. If $a_\ell = 0$, then $a_{\ell,1} = a_{h,0} = 0$ so $s_{\ell,1} = s_{h,0} = b - 1$. So no fixup can occur and $s = a - 1 \bmod b^k$ at line 19.
  If $a_\ell > 0$ and $a_{h,0} = 0$, then $s_{h,0} = b - 1$ and $s_{\ell,1} = a_{\ell,1} = a_{h,0} = 0$ so the first fixup occurs. Thus $s = a - 1$ at line 19.
  If $a_\ell > 0$ and $a_{h,0} > 0$, then no fixup can occur. Thus $s = a - 1$ at line 19.

So the result is proven by induction on the height of tree made of the recursive calls. □

Theorem 5: Assume Algorithm Convert is called with the integer a, then the output string s verifies $s = a$ (identifying s and its value in radix b).

Proof: At line 25 of Algorithm 3, the call of the function CONVERT_REC satisfies Eq. (6) with $d = -3/4$, due to Eq. (1) and line 23 of Algorithm 3. Moreover, line 22 of Algorithm 3 assures us that at most g recursive calls will be made. So Lemma 4 can be applied, and we have $s = a$ or $s = a - 1 \bmod b^k$.

We will now show that the case $s = a - 1 \bmod b^k$ is impossible. Having $s = a - 1 \bmod b^k$ 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 $a - d_0 - g/(4g) < yb^k/2^n < a + 1$, with $d_0 = -3/4$ being the error before the first call to CONVERT_REC, as seen above. Thus, when CONVERT_TRUNC is called, we have $a + 1/2 < yb^k/2^n < a + 1$, which is sufficient to assure that the output of CONVERT_TRUNC will be a, as proven by Theorem 2. So no error can be done on the least significant digit, and so we necessarily have $s = a$. □

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 [5]. 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 that $n_\ell \simeq n - n_\ell \simeq n/2$ and $b^{k-k_\ell} < 2^{n-n_\ell+1}$. 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 left hand side of the inequality about y_ℓ in Lemma 3 and leaves the right hand side 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.

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

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`. In all the following figures, the x-axis represents the number of 64-bit words (limbs) of the corresponding integer or floating-point numbers and the y-axis represents the time (in milliseconds) for one run of the algorithm. 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

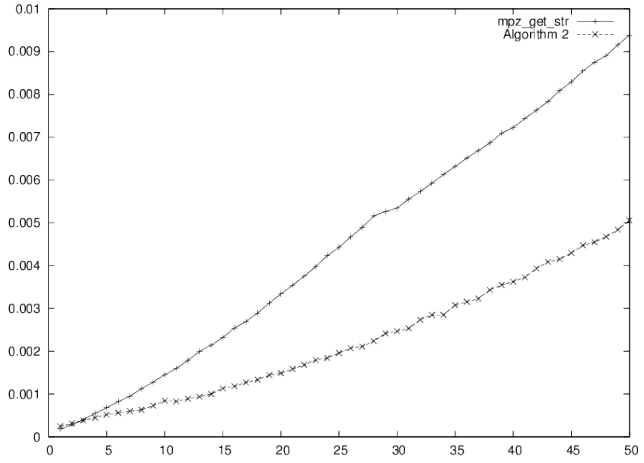

Fig. 1. Integer conversion: comparison with GMP for small sizes

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 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.

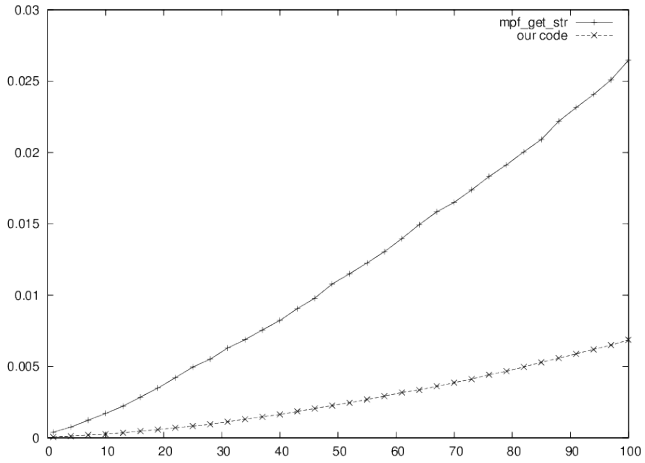

Fig. 2. Floating-point conversion: comparison with GMP for small sizes

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.

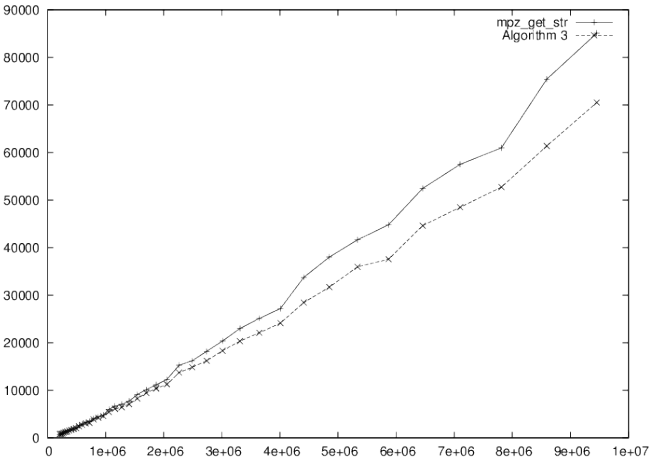

Fig. 3. Integer conversion: comparison with GMP for larger sizes

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.

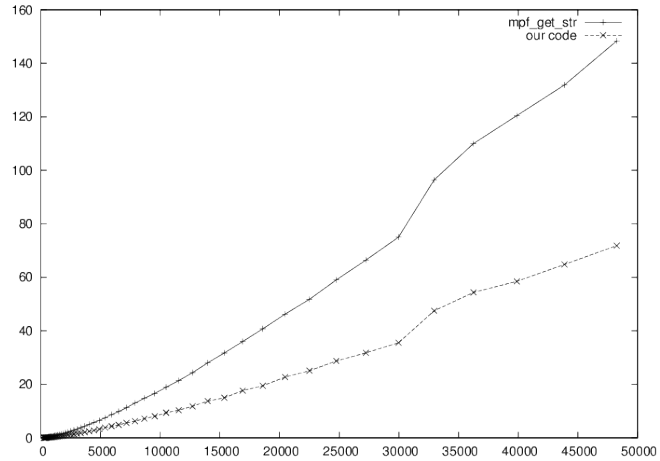

Fig. 4. Floating-point conversion: comparison with GMP for larger sizes

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 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

Both authors thank Alain Filbois who helped them to optimize the implementation of the quadratic algorithm on modern processors. Paul Zimmermann thanks Petr Filipsky who motivated him to work on this problem, by asking an answer to Exercise 1.35 of [2].

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

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

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