# Another Sugar Cube, Please !
# or Sweetening Third Power computation

Alberto Zanoni

Centro “Vito Volterra” – Università di Roma “Tor Vergata”
Via Columbia 2 – 00133 Roma, Italy
zanoni@volterra.uniroma2.it

**Abstract.** A new approach to the computation of long integers cube - third power - computation based on a splitting in two divide et impera approach and on a modified Toom-Cook unbalanced method is presented. A detailed description of its practical implementation by using the GMP library and performance comparison are also described.

*AMS Subject Classification:* 11A05, 11A25, 11K65, 11Y70

*Keywords and phrases:* Polynomial, long integers, multiplication, Karatsuba, Toom

## 1 Introduction

Starting with the works of Karatsuba [6], Toom [10] and Cook [2], who found methods to lower asymptotic complexity for multiplication and squaring from $O(n^2)$ to $O(n^e)$, where $1 < e \leqslant \log_2 3$, many efforts have been done to find optimized implementations in arithmetic software packages [7], [3], [5].

The family of Toom-Cook (Toom, for short) methods is an infinite set of algorithms (Toom-3, Toom-4, etc. – Karatsuba may be identified with Toom-2). The original family was generalized by Bodrato and Zanoni in [1] by considering unbalanced operands as well – that is, polynomials with different degrees – with the so-called Toom-$(k+1/2)$ methods (Toom-2.5, Toom-3.5, etc.) and with the unbalanced use of classic methods.

Each of them may be viewed as solving a polynomial interpolation problem, with base points not specified a priori, from which a matrix to be inverted rises. In a software implementation, a set *Op* of basic operations (typically sums, subtractions, bit shiftings, multiplication and division by small numbers, etc.) is given. Practically, this is a set of very efficiently implemented basic functions in a certain computer language, and the idea is to use them both to evaluate factors in the base points and invert the resulting matrix step by step.

Asymptotically speaking , Shönhage and Strassen’s FFT-based method [8] enters the game when the number of digits is quite large, so that Toom methods have in practice a limited range in which prove to be better. The asymptotically best theoretical algorithm is Fürer’s [4], but as it becomes practical for really huge numbers, it is not realized in practice.

To the best of our knowledge, the problem of computing the cube, $a^3$, has not been deeply treated. In general, a generic power is obtained by a clever sequence of squarings and multiplications, and cube is simply treated as a particular case of this general procedure.

In this paper a new perspective for cube computation is presented in full detail, showing that there does exist a non trivial algorithm, faster (in a certain range) than computing $A = a^2$ and then the product $A \cdot a$. A practical performance comparison with respect to GMP library (release 5.0.1, see [5]) is shown.

## 2 Mathematical setting

In the following, the support of a polynomial $p(x) = \sum_i a_i x^i \in \mathbb{Z}[x]$ is indicated with $supp(p) = \{i \in \mathbb{N} \mid a_i \neq 0\}$. For simplicity, we consider only positive long integers, as taking care of the sign is a standard matter. Consider integers base expansion representation: for $u \in \mathbb{N}$, we fix a base $1 < \mathcal{B} \in \mathbb{N}$ and consider the polynomial determined by the representation of $u$ in base $\mathcal{B}$ with degree $d = \lfloor \log_{\mathcal{B}}(u) \rfloor$ and having as coefficients the digits of $u$, each smaller than $\mathcal{B}$. In computer science, common choices are $\mathcal{B} = 2, 2^8, 2^{16}, 2^{32}, 2^{64}$: in particular, GMP library digits are named *limbs*. Following the divide-and-conquer approach, we set $n = \lceil (d+1)/2 \rceil$, $t = d - n \in \{n-1, n\}$ and $x = y^n$, so that we can write

$$u = \sum_{i=0}^{\lfloor \log_{\mathcal{B}}(u) \rfloor} a_i \mathcal{B}^i \quad \Longrightarrow \quad \hat{a}(y) = \sum_{i=1}^{d} a_i y^i = \left( \sum_{i=0}^{t} a_{n+i} y^i \right) x + \left( \sum_{i=0}^{n-1} a_i y^i \right) = a_1 x + a_0 = a(x) \tag{1}$$

Let $f(x) = a(x)^3$: we can then compute $u^3$ as $f(x)|_{x=\mathcal{B}^n}$. The core idea of the new cube algorithm is based on the following definition.

**Definition 1.** *Let $f(x) = \sum_i a_i x^i$, $g(x) = \sum_i b_i x^i \in \mathbb{Z}[x]$. We say $f$ is a* master *of $g(x)$ (or $f$* masters *$g$) iff supp$(f)$ = supp$(g)$ and $b_i | a_i \; \forall\, i \in$ supp$(f)$. The* authority *(of $f$ with respect to $g$) is the ordered vector $v = [\ a_i/b_i \mid i \in \text{supp}(f)\ ]$. To put in evidence the authority, we say that $g(x)$ is a $v$-master of $f(x)$, or that $g(x)$ $v$-masters $f(x)$.*

*Example 1.* $f(x) = 2x^3 - 4x^2 - 5x + 9$ is a $(2, -2, 1, 3)$-master of $g(x) = x^3 + 2x^2 - 5x + 3$.

Mastery is a reflexive and transitive relation. In the following sections, we will exploit it to present a new approach to cube computation, proving in practice the following

**Theorem 1.** *The cube computation formula $u^3 = (u^2) \cdot u$ is not (always) optimal.*

## 3 On new cube algorithm by splitting in two

To explain the main idea, we'll use both long integers polynomial representation, as explained above, and a schematic one as well. If $u$ is a long integer, we may highlight its high and low half – coefficients $a_1$ and $a_0$ in eq. (1).

$$u = a_1 x + a_0 \;\simeq\; \underbrace{\lfloor\, a_1 \,}_{t}\underbrace{|\, a_0 \,\rfloor}_{n}$$

We indicate with $P_{n,m}$ ($P_n$ if $n = m$) the quadratic complexity of a product of two numbers with $n > m$ digits in base $\mathcal{B}$, respectively; with $S_n$ the squaring complexity of one of them, with $C_n$ the cubing complexity. We must compute (piling means adding)

$$u^3 = f(\mathcal{B}^n) = \sum_{i=0}^{3} c_i \mathcal{B}^{ni} = (a_1\mathcal{B}^n + a_0)^3 = a_1^3x^3 + 3a_1^2a_0x^2 + 3a_1a_0^2x + a_0^3|_{x=\mathcal{B}^n} \;\simeq\; \begin{array}{r} \overset{3t}{a_1^3}\ \big|\ \overset{3n}{a_0^3} \\ 3a_1a_0^2 \ (2n+t) \\ \underset{2t+n}{3a_1^2a_0} \end{array}$$

With the classical "algorithm" $u^3 = u^2 \cdot u$ the quadratic complexity results to be $C_{2n} = S_{2n} + P_{4n,2n}$. Using Karatsuba method to compute $u^2$ (in a range where it's more effective than schoolbook method), we obtain $C_{2n} = 3S_n + P_{4n,2n}$.

Consider now the following very easily factorizable polynomial $g'(x)$, $[1, 1, 1, 9]$-master of $f(x)$:

$$g'(x) = a_1^3x^3 + 3a_1^2a_0x^2 + 3a_1a_0^2x + 9a_0^3 = (a_1^2x^2 + 3a_0^2)(a_1x + 3a_0) = g_1(x) \cdot g_2(x)$$

From a polynomial point of view, $g'(x)$ coefficients computation is similar to $f(x)$'s, as they differ only by a 9 in the constant term: there is nothing particularly appealing in it.

On the contrary, looking from the long integers point of view, things are not as easy but more interesting. Compute $A = a_1^2$ and $B = a_0^2$ (quadratic complexity: $2S_n$) and then focus on the product $g'(\mathcal{B}^n) = (Ax^2 + 3B)(a_1x + 3a_0)|_{x=\mathcal{B}^n}$ (quadratic complexity: $P_{4n,2n}$). Even if this is not really what we want to compute, note that now the total quadratic complexity is now *smaller* than before: one square less gives $C'_{2n} = 2S_n + P_{4n,2n}$.

$$g(\mathcal{B}^n) \simeq \lfloor\, A = a_1^2 \,|\, 3B = 3a_0^2 \,\rfloor \cdot \lfloor\, a_1 \,|\, 3a_0 \,\rfloor$$

There are some points to consider: first, multiplication by 3 may generate carries, to be taken care of (a number possibly differing by 1 or 2 from $\ell$ is below indicated with $\sim\ell$). Second, unfortunately recomposition (setting $x = \mathcal{B}^n$) mixes things up the wrong way. Unbalanced Toom-3 method computing $g'(\mathcal{B}^n)$ splits infact the longest factor in four parts

$$\underbrace{\lfloor\, A_1}_{\sim n}\underbrace{|\, A_0}_{n}\underbrace{|\, 3B_1}_{\sim n}\underbrace{|\, 3B_0 \,\rfloor}_{n} \cdot \underbrace{\lfloor\, a_1}_{n}\underbrace{|\, 3a_0 \,\rfloor}_{\sim n}$$

so that the final division by 9, needed to obtain $u = f(\mathcal{B}^n)$, is incompatible with the recomposition.

$$g'(x) = (A_1a_1)x^4 + (A_0a_1 + 3A_1a_0)x^3 + (3A_0a_0 + 3B_1a_1)x^2 + (3B_0a_1 + 9B_1a_0)x + 9B_0a_0 = \sum_{i=0}^{4} c'_i x^i$$

A problem raises here: $9a_0^3 = 9B_1a_0x + 9B_0a_0|_{x=\mathcal{B}^n}$ cannot be obtained by a linear combination of $c'_i$ coefficients. Infact, it "appears" only after recomposing, while to obtain $b^3$ we should divide by 9 both $c'_0$ (which is not a problem,

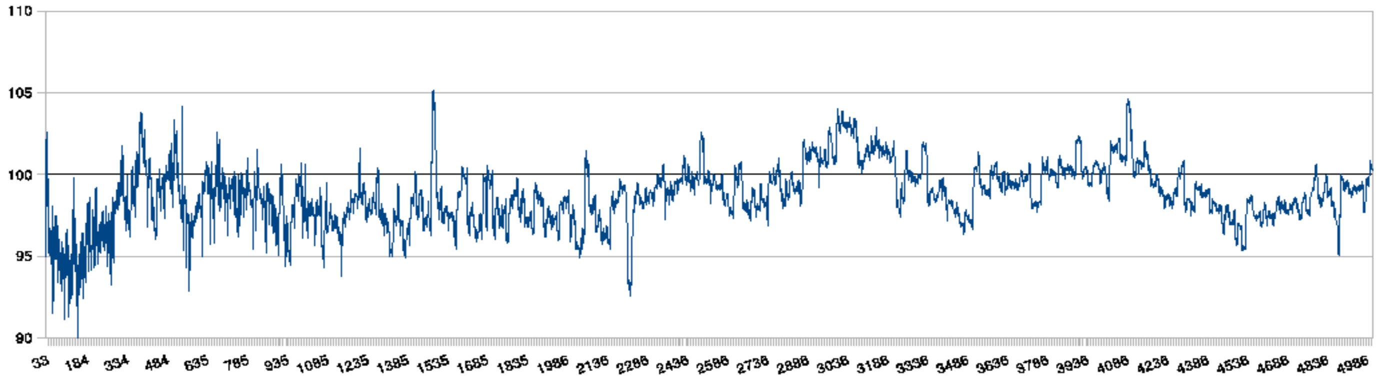


**Fig. 1.** New cube algorithm versus GMP-5.0.1 version relative timings

it is explicitly computed) and only the second addend (which was *not* explicitly computed) summing to $c'_1$. One must therefore proceed in a slightly different way: consider instead the product

$$g(x) = (A_1x^3 + A_0x^2 + 3B_1x + 27B_0)(a_1x + 3a_0) \;\simeq\; \boxed{A_1 \mid A_0 \mid 3B_1 \mid 27B_0} \cdot \boxed{a_1 \mid 3a_0}$$

This way we have

$$g(\mathcal{B}^n) = (A_1a_1)x^4 + (A_0a_1 + 3A_1a_0)x^3 + (3A_0a_0 + 3B_1a_1)x^2 + (27B_0a_1 + 9B_1a_0)x + 81B_0a_0 = \left.\sum_{i=0}^{4} c''_i x^i \right|_{x=\mathcal{B}^n}$$

Note that the coefficients $c''_i$ are correct for $i = 2, 3, 4$. Now we can appropriately divide $c''_1$ by 9 and $c''_0$ by 81 (operations with *linear* complexity), correctly obtaining $f(\mathcal{B}^n) = u^3$.

$$g(\mathcal{B}^n) \;\simeq\; \begin{array}{c} \boxed{A_1a \mid 3B_1a \mid 81B_0b} \\ \boxed{A_0a \mid 27B_0a} \\ \boxed{3A_1b \mid 9B_1b} \\ \boxed{3A_0b} \end{array} \;\Longrightarrow\; \begin{array}{c} \boxed{A_1a \mid 3B_1a \mid B_0b} \\ \boxed{A_0a \mid 3B_0a} \\ \boxed{3A_1b \mid B_1b} \\ \boxed{3A_0b} \end{array} \;= f(\mathcal{B}^n) = u^3$$

We point out that it is possible in practice to avoid the final explicit division by 81, by slightly modifying Toom-3 unbalanced method: compute $B_0b$ instead of $81B_0b$, and in the matrix inversion phase (we consider the optimal inversion presented in [1]) just substitute a subtraction with a multiply-by-81-and-subtract operation. Appendix A contains a high-level implementation of the algorithm in GP-PARI [9], which the reader is invited to refer to.

As (unbalanced) Toom-3 method computes 5 products, the new quadratic complexity is $C'_{2n} = 2S_n + 5P_n < 3S_n + 5P_n = C_{2n}$. A very rough analysis, counting only the number of quadratic operations and forgetting about their nature and linear operations as well, tells that just 7 instead of 8 operations are needed, so that the relative gain cannot be greater than $1/8 = 12.5\,\%$. In figure 1 a relative timings comparison of our `C` implementation with GMP library (release 5.0.1) is shown, for $u$ from 20 to 5000 limbs long. The architecture characteristics for testing are listed below.

| | |
|---|---|
| **Processor** | Intel Core 2 Duo : 3 GHz |
| **RAM** | 4 GB |
| **Operating System** | Linux Kubuntu 8.10 |
| **C Compiler** | gcc 4.3.2 |

## 4 Powers with ternary algorithm

Generic exponentiation $U = (u^e)$ is usually performed by a binary algorithm (in the by side box, let $e_{(2)} = (1, e_k, \ldots, e_0)$ be the representation of the exponent $e$ in base 2).

Ternary exponent expansion $e_{(3)}$ and the corresponding power algorithm was tested to check if the new cube algorithm could be effective in generic exponentiation as well, but the comparison with GMP showed that it is not the case. Anyway, we found that the new algorithm can be effective e.g. for $e \in \{3, 9, 15, 27, 63\}$.

**Binary exponentiation algorithm**

```
let U = u
for i = k to 0 step −1 do
        U ← U^2
        if e_i = 1 then U ← U · u
return U
```

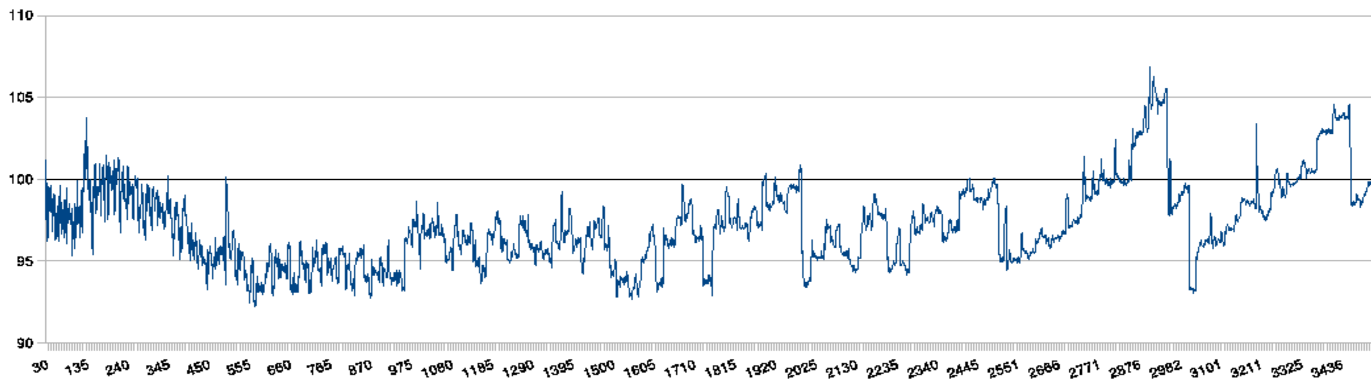


**Fig. 2.** Ninth power by means of new cube algorithm versus GMP-5.0.1 version relative timings

As an example, in figure 2 the result of the comparison with GMP for exponent $e = 9$ is shown.

## 5 Conclusions

In this work the classical algorithm to compute the third power of a long integer was proved not to be (always) optimal. A new algorithm using a combination of Karatsuba and unbalanced Toom-3 methods paying some linear operations more for a quadratic operation less was proposed. It proved to be effective also in practice: by comparison with GMP library the obtained saving was up to around 8 %.

## 6 Acknowledgments

The author thanks Marco Bodrato for many helpful discussions and hints. This work is dedicated to the author's beloved daughter, Marina Zanoni.

## References

1. Marco Bodrato and Alberto Zanoni. Integer and polynomial multiplication: Towards optimal Toom-Cook matrices. In Christopher W. Brown, editor, *Proceedings of the ISSAC 2007 Conference*. ACM press, July 2007.
2. Stephen A. Cook. *On the minimum computation time of functions*. PhD thesis, Department of Mathematics, Harvard University, 1966.
3. Tom St Denis, Mads Rasmussen, and Greg Rose. Multi-precision math (tommath library documentation). `http://math.libtomcrypt.com/files/tommath.pdf`.
4. Martin Fürer. Faster integer multiplication. In David S. Johnson and Uriel Feige, editors, *STOC*, pages 57–66. ACM, 2007.
5. Torbjörn Granlund. The GNU multiple precision (GMP) library. `http://gmplib.org/`.
6. Anatolii Alexeevich Karatsuba and Yuri Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7(7):595–596, 1963.
7. Donald E. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*. Addison-Wesley, 1981.
8. A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7(3–4):281–292, 1971.
9. The PARI Group, Bordeaux. *PARI/GP, version* `2.3.4`, 2008. available from `http://pari.math.u-bordeaux.fr/`.
10. Andrei L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics Doklady*, 3:714–716, 1963.

## A GP-PARI code for cube computation

We report here a GP-PARI high-level implementation of the new algorithm for cube computation.

```
cube(a = 2) =
{
 local(a0,a1,A0,A1,A00,A01,A10,A11,A00orig,half,tmp);

 \\ Compute a0 and a1.
 half = 2^(ceil(log(a)/log(2)+1)>>1);
 tmp  = divrem(a,half);
 a1   = tmp[1]; a0 = tmp[2];

 \\ Identify high and low parts of a1^2 and a0^2.
 A1  = a1^2;   A0      = a0^2;           \\ 2S(n)
 tmp = divrem(A0,half);
 A01 = tmp[1]; A00orig = tmp[2];
 tmp = divrem(A1,half);
 A11 = tmp[1]; A10     = tmp[2];

 A01 *= 3; A00 = 27*A00orig;

 \\ Unbalanced Toom-3 (toom42) in (oo, 2, 1, -1, 0).
 \\ of AA = A11*x^3 + A10*x^2 + A01*x + A00;
 \\    BB = a1*x + (3*a0);
 \\ Evaluation:
 \\     AA                    BB
 W2   = A10 + A00;
 Wm1  = A11 + A01;
 W0   = W2 + Wm1;
 W2   = W2 - Wm1;       Wm1  = 3*a0;
                        Winf = Wm1 + a1;
 W1   = W0*Winf;    \\ Evaluation in 1       P(n)
                        Winf = Winf + a1;
                        W0   = Wm1  - a1;
 Wm1  = W0*W2;      \\ Evaluation in -1      P(n)
 W0   = A11<<1 + A10;
 W0   = W0<<1  + A01;
 W0   = W0<<1  + A00;
 W2   = W0*Winf;    \\ Evaluation in 2       P(n)
 Winf = A11*a1;     \\ Evaluation in oo      P(n)
 W0   = A00orig*a0; \\ Evaluation in 0 (/81) P(n)

 \\ Interpolation:                       Winf [ 1  0  0  0  0]
                     \\                  W2   [16  8  4  2  1]
 W2  = W2 - Wm1;     \\ [15 9 3 3 0]     W1   [ 1  1  1  1  1]
 W2 /= 3;            \\ [ 5 3 1 1 0]     Wm1  [ 1 -1  1 -1  1]
 Wm1 = (W1 - Wm1)>>1; \\ [ 0 1 0 1 0]
 W1  = W1 - 81*W0;   \\ [ 1 1 1 1 0]
 W2  = (W2 - W1)>>1;  \\ [ 2 1 0 0 0]
 W1  = W1 - Wm1;     \\ [ 1 0 1 0 0]
 W2  = W2 - Winf<<1;  \\ [ 0 1 0 0 0]
 W1  = W1 - Winf;    \\ [ 0 0 1 0 0]
 Wm1 = Wm1 - W2;     \\ [ 0 0 0 1 0]
                     \\
 Wm1 /= 9;           \\ Extra division.

 \\ Recomposition:
 return ((((Winf*half + W2)*half + W1)*half + Wm1)*half + W0);
}
```