On Newton-Raphson iteration for multiplicative inverses modulo prime powers

Jean-Guillaume Dumas

▶ To cite this version:

Jean-Guillaume Dumas. On Newton-Raphson iteration for multiplicative inverses modulo prime powers. 2012. hal-00736701v2

HAL Id: hal-00736701
https://hal.science/hal-00736701v2

Submitted on 2 Oct 2012 (v2), last revised 14 May 2018 (v5)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

On Newton-Raphson iteration for multiplicative inverses modulo prime powers

Jean-Guillaume Dumas*

October 2, 2012

Abstract

We study algorithms for the fast computation of modular inverses. We first give another proof of the formulas of [1] for the modular inverse modulo 2^m, derived from Newton-Raphson iteration over p-adic fields, namely Hensel's lifting. From the expression of Newton-Raphson's iteration we then derive an actually explicit formula for the modular inverse, generalizing to any prime power modulus. On the one hand, we show then that despite a worse complexity the explicit formula can be 4 times faster than Arazi and Qi's for small exponents. On the other hand, this algorithm becomes slower for arbitrary precision integers of more than 1700 bits. Overall a hybrid combination of the two latter algorithms yield a constant factor improvement also for large exponents.

1 Introduction

The multiplicative inverse modulo a prime power is fundamental for the arithmetic of finite rings (see e.g. [1] and references therein). It is also used for instance to compute homology groups in algebraic topology for image pattern recognition [4].

Classical algorithms to compute a modular inverse uses the extended Euclidean algorithm and [1] lists also some variants, linear in the power of the prime, adapted to the binary characteristic case. Arazi and Qi also present in [1] a method working in characteristic 2 that has a complexity logarithmic in the prime power.

In the following, we give another proof of Arazi and Qi's logarithmic formula using classical Newton-Raphson iteration over p-adic fields. The latter is usually called Hensel lifting. Then from this variation we derive the classical Newton-Raphson formula and an explicit formula for the inverse that generalizes to any prime power. Finally, we study the respective performance of the three

*Université de Grenoble; Laboratoire Jean Kuntzmann, (umr CNRS 5224, Grenoble INP, INRIA, UJF, UPMF); 51, rue des Mathématiques, BP 53X, F-38041 Grenoble, France. Jean-Guillaume.Dumas@imag.fr

algorithms both asymptotically and in practice and introduce a hybrid algorithm combining both approaches.

2 Hensel's lemma modulo p^m

For the sake of completeness, we first give here Hensel's lemma and its proof from Newton-Raphson's iteration (see e.g. [2, Theorem 7.7.1] and references therein).

Lemma 1 (Hensel)**.** *Let p be a prime number, $m \in \mathbb{N}$, $f \in \mathbb{Z}[X]$ and $r \in Z$ such that $f(r) = 0 \mod p^m$. If $f'(r) \neq 0 \mod p^m$ and*

$$t = -\frac{f(r)}{p^m} f'(r)^{-1},$$

then $s = r + tp^m$ satisfies $f(s) = 0 \mod p^{2m}$.

Proof. Taylor expansion gives that $f(r+tp^m) = f(r)+tp^m f'(r)+O(p^{2m})$. Thus if $t = -\frac{f(r)}{p^m} f'(r)^{-1}$, the above equation becomes $f(s) = 0 \mod p^{2m}$. □

3 Inverse modulo 2^m

Now, in the spirit of [7], we apply this lemma to the inverse function

$$F_a(x) = \frac{1}{ax} - 1 \tag{1}$$

3.1 Arazi and Qi's formula

We denote by an under-script $_L$ (resp. $_H$) the lower (resp. higher) part in binary format for an integer. From Equation (1) and Lemma 1 modulo 2^i, if $r = a^{-1} \mod 2^i$, then we immediately get

$$t = -\frac{\frac{1}{ax} - 1}{2^i} \left(-\frac{1}{ax^2} \right)^{-1}.$$

In other words $t = \frac{1-ar}{2^i} r \mod 2^i$. Now let $a = b + 2^i a_H \mod 2^{2i}$ so that we also have $r = b^{-1} \mod 2^i$ and hence $rb = 1 + 2^i\alpha$ with $0 \leq \alpha < 2^i$. Thus $ar = br + 2^i r a_H = 1 + 2^i(\alpha + r a_H)$ which shows that

$$t = -(\alpha + r a_H) r \equiv -\left((rb)_H + (r a_H)_L \right) r \mod 2^i \tag{2}$$

The latter is exactly [1, Theorem 1] and yields the following Algorithm 1, where the lower and higher parts of integers are obtained via masking and shifting.

Lemma 2. *Algorithm 1 requires $13\lfloor \log_2(m) \rfloor$ arithmetic operations.*

Algorithm 1 Hensel Quadratic Modular inverse

Input: $a \in \mathbb{Z}$ odd and $m \in N$
Output: $U \equiv a^{-1} \mod 2^m$

1: $U = 1$;
2: **for** $(i = 1;\ i < m;\ i <<= 1)$ **do**
3: $\quad b = a \;\&\; (2^i - 1)$; $\{b = a \mod 2^i\}$
4: $\quad t_1 = U * b;\ t_1 >>= i$; $\{(rb)_H\}$
5: $\quad c = (a >> i) \;\&\; (2^i - 1)$; $\{a_H\}$
6: $\quad t_2 = (U * c) \;\&\; (2^i - 1)$; $\{(ra_H)_L\}$
7: $\quad t_1 += t_2$;
8: $\quad t_1 *= U;\ t_1 \;\&= \;(2^i - 1)$; $\{-t\}$
9: $\quad t_1 <<= i$; $\{-t_2^i\}$
10: $\quad U| = 2^i - t_1$; $\{r + t_1 2^i\}$
11: $\quad U \;\&= \;(2^{2i} - 1)$; $\{r \mod 2^{2i}\}$
12: **end for**
13: **return** U;

3.2 Recurrence formula

Another view of Newton-Raphson's iteration is to create a recurrence. Equation (1) gives

$$U_{n+1} = U_n - \frac{\frac{1}{aU_n} - 1}{-\frac{1}{aU_n^2}} = U_n - (aU_n - 1)U_n = U_n(2 - aU_n)$$

This yields the recursive loop of Algorithm 2, for the computation of the inverse, see e.g. [3, §2.4].

Algorithm 2 Recursive Quadratic Modular inverse

Input: $a \in \mathbb{Z}$ odd, p is a prime and $m \in N$
Output: $U \equiv a^{-1} \mod p^m$

1: $U = a^{-1} \mod p$; $\{\text{extended gcd}\}$
2: **for** $(i = 0;\ i < m;\ i <<= 1)$ **do**
3: $\quad temp = 2 - a * U$; $\{2 - aU_n\}$
4: $\quad temp \;\%= \;p^m$; $\{temp \mod p^m\}$
5: $\quad U *= temp$; $\{U_n(2 - aU_n)\}$
6: $\quad U \;\%= \;p^m$; $\{U \mod p^m\}$
7: **end for**
8: **return** U;

This algorithm can e.g. improve the running time of algorithms working modulo prime powers. Those can be used for the computation of the local Smith normal form [5, 6], for instance in the context of algebraic topology [4, algorithm LRE].

Lemma 3. *Algorithm 2 is correct and requires* $5\lceil\log_2(m)\rceil + 1$ *arithmetic operations.*

Proof. The proof of correctness is natural in view of the Hensel lifting. First $U_0 = a^{-1} \mod p$. Second, by induction, suppose $a \cdot U_n \equiv 1 \mod p^k$. Then $aU_n = 1+\lambda p^k$ and $aU_{n+1} = aU_n(2-aU_n) = (1+\lambda p^k)(2-1-\lambda p^k) = (1-\lambda^2 p^{2k} \equiv 1 \mod p^{2k})$. Finally $U_n \equiv a^{-1} \mod p^{2^n}$. □

Remark 1. *We present this algorithm for computations modulo* p^m *but its optimization modulo a power of* 2 *is straightforward: replace the modular operations of lines 4 and 6 by a binary masking:* $x \mathrel{\&}= (2^m - 1)$.

3.3 Factorized formula

We now give an explicit formula for the inverse by solving the preceding recurrence relation, first in even characteristic.

We denote by $H_n = aU_n$ a new sequence, that satisfies $H_{n+1} = H_n(2 - H_n)$. With $H_0 = a$ we get $H_1 = a(2-a) = 2a - a^2 = 1 - (a-1)^{2^1}$, by induction, supposing that $H_n = 1 - (a-1)^{2^i}$, we get

$$\begin{aligned} H_{n+1} &= \left(1 - (a-1)^{2^n}\right)\left(2 - 1 + (a-1)^{2^n}\right) \\ &= 1^2 - \left((a-1)^{2^n}\right)^2 \\ &= 1 - (a-1)^{2^{n+1}} \end{aligned}$$

Using the remarkable identity, this in turn yields

$$H_n = a(2-a)\prod_{i=1}^{n-1}\left(1 + (a-1)^{2^i}\right)$$

Therefore, with $U_0 = 1$ and $U_1 = 2 - a$ we have that

$$U_n = (2-a)\prod_{i=1}^{n-1}\left(1 + (a-1)^{2^i}\right) \tag{3}$$

Lemma 4. *Algorithms 3 is correct and requires* $5\lfloor\log_2(\frac{m}{s})\rfloor + 2$ *arithmetic operations.*

Proof. Modulo 2^m, a is invertible if and only if a is odd, so that $a = 2^s t + 1$ and therefore, using Formula (3), we get

$$aU_n = H_n = 1 - (a-1)^{2^n} = 1 - (2^s t)^{2^n} \equiv 1 \mod 2^{s2^n}$$

Therefore,

$$U_{\lceil\log_2(\frac{m}{s})\rceil} \mod 2^m \equiv a^{-1} \mod 2^m$$

□

Algorithm 3 Explicit Quadratic Modular inverse modulo 2^m

Input: $a \in \mathbb{Z}$ odd and $m \in N$
Output: $U \equiv a^{-1} \mod 2^m$
1: Let s and t be such that $a = 2^s t + 1$;
2: $U = 2 - a$;
3: $amone = a - 1$;
4: **for** $(i = 1;\ i < \frac{m}{s};\ i <<= 1)$ **do**
5: $\quad amone *= amone$; $\quad$ {square: $(a-1)^{2^i}$}
6: $\quad amone \mathrel{\&}= (2^m - 1)$; $\quad$ {$(a-1)^{2^i} \mod 2^m$}
7: $\quad U *= (amone + 1)$;
8: $\quad U \mathrel{\&}= (2^m - 1)$; $\quad$ {$U \mod 2^m$}
9: **end for**
10: **return** U;

3.4 Generalization modulo any prime power

The formula generalizes directly for any prime power as follows:

Theorem 1. *Let p be a prime number, a coprime to p and $b = a^{-1} \mod p$ is the inverse of a modulo p. Let also V_n be the following sequence:*

$$\begin{cases} V_0 & = b \equiv a^{-1} \mod p, \\ V_n & = b(2 - ab) \prod_{i=1}^{n-1} \left(1 + (ab-1)^{2^i}\right) \end{cases} \tag{4}$$

Then

$$V_n \equiv a^{-1} \mod p^{2^n}. \tag{5}$$

Proof. The proof is similar to that of Lemma 3 and follows also from Hensel's lemma. From the analogue of Equation (3), we have $a \cdot V_n = 1 - (ab-1)^{2^n}$. Now as $a \cdot b = 1 + \lambda p$, by the definition of b we have $a \cdot V_n = 1 - (ab-1)^{2^n} = 1 - (\lambda p)^{2^n} \equiv 1 \mod p^{2^n}$. □

4 Experimental comparisons

The point of the algorithm Arazi and Qi is that it works with modular computations of increasing sizes, whereas the explicit formula requires to work modulo the highest size from the beginning. On the hand we show next that this gives an asymptotic advantage to Arazi and Qi's algorithm. On the other hand, in practice, the explicit formula enables much faster performance for say cryptographic sizes.

4.1 Over word-size integers

Using word-size integers, the many masking and shifting required by Arazi and Qi's algorithm do penalize the performance, where the simpler Algorithm 3 can as much as 4 times faster on a standard desktop PC, as shown on Figure 1.

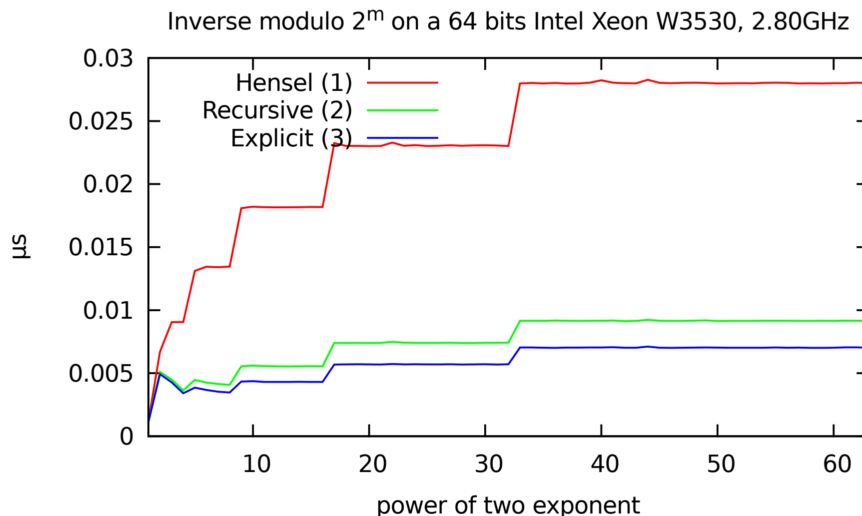

Figure 1: Modular inverse on 64 bits machine words

With regards to Algorithm 2, Algorithm 3 uses about the same number of arithmetic operations. However it replaces, first, one multiplication by a squaring and, second, one subtraction by a simple increment. These can yield some constant factor improvements in practice [8]. In our case the improvement is of about 30%.

4.2 Over arbitrary precision arithmetic

We first provide the equivalents of the complexity results of the previous section but now for arbitrary precision: the associated binary complexity bounds for the different algorithms supports then the asymptotic analysis in the beginning of this section.

Lemma 5. *Using classical arithmetic, Algorithm 1 requires*

$$\mathcal{O}\left(4m^2 + 20m\right) \quad \textit{binary operations.}$$

Proof. We suppose that masking and shifting as well as addition are linear and that multiplication is quadratic. Then the complexity bound becomes $\mathcal{O}\left(\sum_{j=1}^{\log_2(m)} 3(2^j)^2 + 10(2^j)\right) = \mathcal{O}\left(4m^2 + 20m\right)$. □

Lemma 6. *Using classical arithmetic, Algorithm 3 requires*

$$\mathcal{O}\left(\left(2m^2 + 3m\right)\lfloor\log_2(m)\rfloor\right) \quad \textit{binary operations.}$$

Proof. Similarly, here we have $\mathcal{O}\left(\sum_{j=1}^{\log_2(m)} 2m^2 + 2m\right) = \mathcal{O}\left(2m^2 \log_2(m) + 2m \log_2(m)\right)$. □

Thus, we see that the explicit formula adds a logarithmic factor, asymptotically. Now, in [7], the recurrence relation from (3), $V_{n+1} = V_n(1 + (1 - ab)^{2^n})$, is replaced by $X_{n+1} = \frac{1-(1-aX_n)^r}{a}$ for a fixed r. This allows a faster convergence, by a factor of $\log_2(r)$. Unfortunately the price to pay is to compute a r-th power at each iteration (instead of a single square), which could be done, say by recursive binary squaring, but at a price of $\log_2(r)$ squarings and between 0 and $\log_2(r)$ multiplications. Overall there would be no improvement in the running time.

In practice, Figure 2 shows that using GMP[1] the asymptotic behavior of Algorithm 1 becomes predominant only for integers with more than 1700 bits. Below that size, Algorithm 3 is better. Its improvement over Algorithm 2 is of about 25% asymptotically, which, with 2 multiplications, matches the 50% gain of squaring over multiplication in an arbitrary precision setting, as reported in [8].

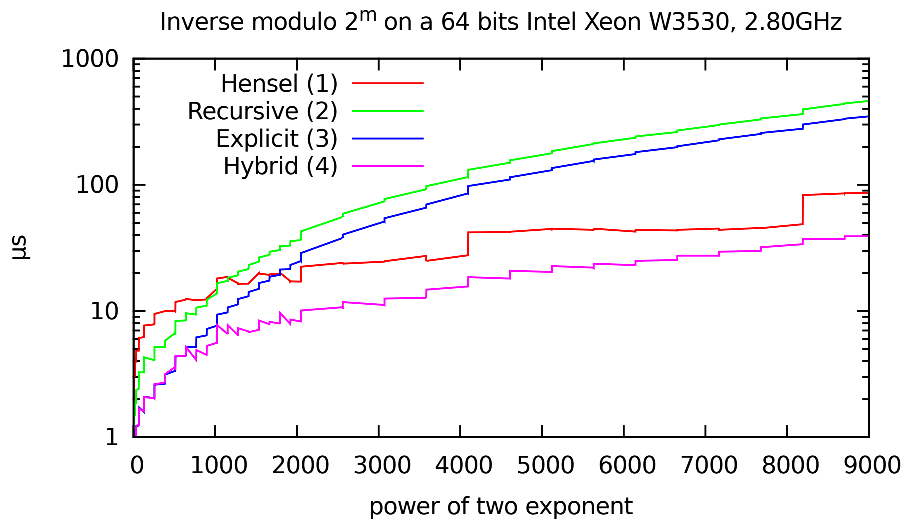

Figure 2: Modular inverse on arbitrary precision integers

4.3 Hybrid algorithm

With the threshold of Figure 2, we can then construct a hybrid algorithm from the previous ones, which will be better than both :

[1] http://gmplib.org

Algorithm 4 Hybrid Modular inverse modulo 2^m

Input: $a \in \mathbb{Z}$ odd and $m \in N$
Output: $U \equiv a^{-1} \mod 2^m$
1: **if** $m < \frac{1700}{2}$ **then**
2: **return** Explicit(a, m); {Algorithm 3}
3: **else**
4: Let $h = \lceil \frac{m}{2} \rceil$;
5: Let $b \equiv a \mod 2^h$;
6: r =Hybrid(b, h); {recursive call}
7: Compute t from a, b, r and Equation (2);
8: **return** $r + t \cdot p^h$. {Correct from Lemma 1}
9: **end if**

Finally, Algorithm 4 is on average 2.47 times faster than Arazi and Qi's version (recall that on Figure 2 ordinates are presented in a logarithmic scale).

5 Conclusion

We have studied different variants of Newton-Raphson's iteration over p-adic numbers to compute the inverse modulo a prime power. We have shown that the classical recurrence relation as well as a new explicit formula can be up to 4 times faster in practice than the version of Arazi and Qi's. Asymptotically, though, the latter formula gains a logarithmic factor in the power (or a doubly logarithmic factor in the prime power) that makes it faster for large arbitrary precision integers. Then, using each one of the best two algorithms in their respective regions of efficiency, we were able to make a hybrid algorithm with improved performance at any precision.

A generalization of Arazi and Qi's formula, requiring only increasing remaindering, could naturally be derived, as we derived generalizations to our explicit formula. Unfortunately, there, the computations of the high and low parts modulo p^α would require quite a lot of computing effort.

References

[1] O. Arazi and Hairong Qi. On calculating multiplicative inverses modulo 2^m. *IEEE Transactions on Computers*, 57(10):1435–1438, October 2008.

[2] Eric Bach and Jeffrey Shallit. *Algorithmic Number Theory: Efficient Algorithms*. MIT press, 1996.

[3] Richard P. Brent and Paul Zimmermann. *Modern computer arithmetic*, volume 18 of *Cambridge monographs on applied and computational mathematics*. Cambridge University Press, Cambridge, UK, 2011.

[4] Jean-Guillaume Dumas, Frank Heckenbach, B. David Saunders, and Volkmar Welker. Computing simplicial homology based on efficient Smith normal form algorithms. In Michael Joswig and Nobuki Takayama, editors, *Algebra, Geometry and Software Systems*, pages 177–206. Springer, 2003.

[5] Jean-Guillaume Dumas, B. David Saunders, and Gilles Villard. On efficient sparse integer matrix Smith normal form computations. *Journal of Symbolic Computation*, 32(1/2):71–99, July–August 2001.

[6] Mustafa Elsheikh, Mark Giesbrecht, Andy Novocin, and B. David Saunders. Fast computation of Smith forms of sparse matrices over local rings. In Joris van der Hoeven and Mark van Hoeij, editors, *ISSAC'2012, Proceedings of the 2012 ACM International Symposium on Symbolic and Algebraic Computation, Grenoble, France*, pages 146–153. ACM Press, New York, July 2012.

[7] Michael Knapp and Christos Xenophontos. Numerical analysis meets number theory: using rootfinding methods to calculate inverses mod p^n. *Applicable Analysis and Discrete Mathematics*, 4(1):23–31, 2010.

[8] Dan Zuras. More on squaring and multiplying large integers. *IEEE Transactions on Computers*, 43(8):899–908, August 1994.