NEWTON ITERATION REVISITED

GUILLAUME HANROT AND PAUL ZIMMERMANN

Abstract. On March 10, 2004, Dan Bernstein announced a revised draft of his paper "Removing redundancy in high-precision Newton iteration" (`http://cr.yp.to/fastnewton.html`) with algorithms that compute a reciprocal of order n over $\mathbb{C}[[x]]$ $1.5 + o(1)$ times longer than a product; a quotient or logarithm $2.16666... + o(1)$ times longer; a square root $1.83333... + o(1)$ times longer; an exponential $2.83333... + o(1)$ times longer. We give better algorithms.

1. Model of computation

All complexities are expressed in terms of $M(n)$, the time needed to multiply two polynomials of degree $< n$. We assume $M(kn) \sim kM(n)$ for any constant k. We also assume that the multiplication algorithm used to achieve $M(n)$ performs a cyclic convolution, i.e. to compute the product of two polynomials of degree $< n$, it computes their product modulo $x^{2n} - 1$. We thus assume that we can compute a product modulo $x^{kn} - 1$ in $\frac{k}{2}M(n)$ for any constant k.

In the cost analysis of each operation, we consider the cost to go from an approximation of order n to an approximation of order $2n$. Since $M(2n) \sim 2M(n)$, this will give exactly the cost to get an approximation of order n.

We also consider that the $M(n)$ cost splits in three halves of $\frac{1}{3}M(n)$ for each FFT transform (two directs and one inverse), the term-to-term multiplication having negligible cost wrt $M(n)$.

2. Reciprocal

We use the classical iteration $g = g_0 + g_0(1 - fg_0)$, that we compute as $g = g_0 + (g_0 - fg_0^2)$. If g_0 is an approximation of order n of $1/f$, then the n most significant coefficients of fg_0^2 cancel with those of g_0, thus we need only to compute the coefficients of order n to $2n-1$. We do this by computing $fg_0^2 \bmod (x^{3n} - 1)$, whose n middle coefficients are what we want. This costs $1.5M(n)$ according to our model of computation.

This algorithm is more efficient than the $2M(n)$ algorithm proposed by Bernstein with the $x^n - 1$ trick, and as efficient — but much simpler — than his "messy" algorithm.

Note that Bernstein "FFT" optimization — i.e. caching the FFT evaluation of some variables that are used twice — does not apply here, since both f and g_0 are used only once.

3. Quotient

We propose the following algorithm, using Karp-Markstein's trick [2] (which is incorrectly used by Bernstein):

Algorithm Divide$(h, f, 2n)$.
1. $g_0 \leftarrow \text{Reciprocal}(f, n)$ [cost $1.5M(n)$]
2. $q_0 \leftarrow hg_0 \bmod x^n$ [cost $M(n)$]
3. $\epsilon \leftarrow \text{MiddleProduct}(q_0, f)$ [cost $M(n)$]
4. $q \leftarrow q_0 - g_0\epsilon$ [cost $M(n)$]

The total cost is therefore $4.5M(n)$ for precision $2n$, or $2.25M(n)$ for precision n. The `MiddleProduct` operation at step 3 computes the n middle coefficients of the size-$3n$ product $q_0 f$ (q_0 has n terms and f has $2n$ terms, see [1]), which could also be computed by a product modulo $x^{2n} - 1$.

If we cache FFT transforms, we gain $1/3M(n)$ by saving the transform of g_0 between steps 2 and 4, which gives $2.0833...M(n)$, which is better than Bernstein FFT and "messy" algorithms.

Date: 22 March 2004.

4. Square Root

To compute a square root on $2n$ bits, we use the following subroutine which computes an inverse square root of order n in $2M(n)$:

Algorithm InverseSquareRoot$(h, 2n)$.
1. $g_0 \leftarrow$ InverseSquareRoot(h, n)
2. $g = g_0 + \frac{1}{2}(g_0 - hg_0^3)$

At step 2, hg_0^3 has $5n$ bits, and we want only bits n to $2n$ — the low n bits are known to match those of g_0 —, thus we can compute $hg_0^3 \bmod x^{4n} - 1$ which costs $2M(n)$.

We then use the following algorithm for the square root:

Algorithm SquareRoot$(h, 2n)$.
1. $g_0 \leftarrow$ InverseSquareRoot(h, n) [cost $2M(n)$]
2. $f_0 \leftarrow hg_0 \bmod x^n$ [cost $M(n)$]
3. $\epsilon \leftarrow h - f_0^2$ [cost $0.5M(n)$]
4. $f \leftarrow f_0 + \frac{1}{2}g_0\epsilon$ [cost $M(n)$]

Since the n most significant bits of f_0^2 are known to match those of h in step 3, we can do a transform mod $x^n - 1$. The total cost is thus $4.5M(n)$ for order $2n$, and thus $2.25M(n)$ for order n.

If we cache the transform of g_0 between steps 2 and 4, and the transform of h at step 2 with InverseSquareRoot, we save $2/3M(n)$ for order $2n$, which gives $1.91666...M(n)$ for order n.

5. Exponential

We use the following algorithm, which computes simultaneously $f = \exp h$ up to order $2n$, and $g = \exp(-h)$ up to order n:

Algorithm Exp$(h, 2n)$.
1. $(f_0, g_0) \leftarrow$ Exp$(h, n+1)$
2. $g \leftarrow g_0 + (g_0 - f_0 g_0^2)$ [cost $0.75M(n)$]
3. $q \leftarrow h' \bmod x^n$
4. $q \leftarrow q + g(f_0' - f_0 q)$ [cost $1.5M(n)$]
5. $f \leftarrow f_0 + f_0(h - \int q)$

After step 1, we have $f_0 = \exp h + O(x^n)$ and $g_0 = \exp(-h) + O(x^{n/2})$; after step 2, we have $g = \exp(-h) + O(x^n)$; after step 3, we have $q = f'/f + O(x^n)$; after step 4, we have $q = f'/f + O(x^{2n})$; finally after step 5, we have $f = \exp h + O(x^{2n})$.

Step 2 costs $0.75M(n)$ (see Reciprocal); step 4 costs $1.5M(n)$ ($f_0 q$ costs $0.5M(n)$ since it has $2n$ coefficients only, and the n most significant are known to match f_0', so we can compute it modulo $x^n - 1$); step 5 costs $M(n)$. The total cost is therefore $3.25M(n)$ for the exponential up to order n.

References

[1] Hanrot, G., Quercia, M., and Zimmermann, P. The middle product algorithm, I. Speeding up the division and square root of power series. *AAECC 14*, 6 (2004), 415–438.

[2] Karp, A. H., and Markstein, P. High-precision division and square root. *ACM Trans. Math. Softw. 23*, 4 (1997), 561–589.