Target: Mathematics of Computation. New version of paper in preparation, with 4.5 for reciprocal, 6.5 for quotient, 5.5 for square root, and 8.5 for exponential. See http://cr.yp.to/fastnewton.html.

REMOVING REDUNDANCY IN HIGH-PRECISION NEWTON ITERATION

DANIEL J. BERNSTEIN

ABSTRACT. This paper speeds up Brent’s algorithms for various high-precision computations in the power series ring $\mathbb{C}[[t]]$. If it takes time $3$ to compute a product then it takes time roughly $5.6$ to compute a reciprocal; roughly $8.2$ to compute a quotient or a logarithm; roughly $6.5$ to compute a square root; roughly $9$ to compute both a square root and a reciprocal square root; and roughly $10.4$ to compute an exponential. The same ideas apply to approximate computations in $\mathbb{R}, \mathbb{Q}_p,$ etc.

1. INTRODUCTION

Let $f \in \mathbb{C}[[t]]$ be a power series with constant coefficient $1$. How can one compute $f^{-1}$? The standard answer is Newton’s method, which shows how to compute $f^{-1} \bmod t^{2n}$ from $f^{-1} \bmod t^n$ with a few size-$n$ multiplications: $f^{-1} \bmod t^{2n} = g_0 + (1 - f g_0) g_0 \bmod t^{2n}$ where $g_0 = f^{-1} \bmod t^n$. One can compute $g_0$ by the same method recursively. (This algorithm is sometimes credited to Sieveking, who published it in [8]; Kung in [6] pointed out that Sieveking’s method was an example of Newton’s method.) Similar comments apply to computing $f^{1/2}$, $\log f$, et al.

Using FFTs one can multiply polynomials of degree up to $n$ with $3kn \log n + O(n)$ arithmetic operations in $\mathbb{C}$ for some constant $k$. Brent in [5] showed that one can compute the first $n$ coefficients of $f^{-1}$ with $9kn \log n + O(n)$ operations, $\log f$ with $12kn \log n + O(n)$ operations, $f^{-1/2}$ with $13.5kn \log n + O(n)$ operations, both $f^{1/2}$ and $f^{-1/2}$ with $16.5kn \log n + O(n)$ operations, and $\exp f$ with $22kn \log n + O(n)$ operations.

The point of this paper is that some obvious redundancies account for a large fraction of the run time of Brent’s algorithms. Sections 2, 3, 4, and 5 present several streamlined examples of Newton’s method, culminating in an algorithm to compute $n$ coefficients of $\exp f$ with only $10.4kn \log n + O(n)$ operations. Section 6 gives implementation results.

Generalizations. Newton’s method is not limited to $\mathbb{C}[[t]]$; it is also used for high-precision computations in $k[[t]]$ where $k$ is a finite field, in the $p$-adic numbers $\mathbb{Q}_p$, in $\mathbb{R}$, and so on. See [3] for a survey of relevant multiplication methods. Roundoff error analysis is typically required for $\mathbb{Q}_p$, where it is easy, and for $\mathbb{R}$, where it is not so easy; see, e.g., [2, section 8 and section 21]. Some functions, such as $\log$, require completely different methods for $\mathbb{R}$; see, e.g., [5].

Brent’s methods can be streamlined in all of these situations. I have focused on $\mathbb{C}[[t]]$ in this paper since it is the simplest case. It is also the case used in [4].

Date: DRAFT 19980627.
1991 Mathematics Subject Classification. Primary 68Q40; Secondary 65Y20.
The author was supported by the National Science Foundation under grant DMS-960083.
Notation and terminology. Throughout this paper, $n$ is a positive integer. Subscripted variables such as $f_0, f_1, \ldots$ refer to polynomials of degree below $n$. Thus every power series can be written uniquely in the form $f_0 + t^n f_1 + t^{2n} f_2 + \cdots$.

A transform means a size-2n FFT or a size-2n inverse FFT. If $p$ is a polynomial of degree below $2n$ then $p^*$ means the result of applying a size-2n FFT to $p$. Recall that one can compute a bilinear form such as $f_0 g_2 + f_1 g_1 + f_2 g_0$, given $f_0^*, f_1^*, f_2^*, g_0^*, g_1^*, g_2^*$, with a single transform plus $O(n)$ operations.

If $f$ is a power series then $D(f)$ means $t$ times the derivative of $f$. If $f$ is a power series with constant coefficient 0 then $I(f)$ means the integral of $f/t$.

2. Reciprocals

Let $f$ and $g$ be power series with $fg = 1$. This section considers the problem of computing $g$ given $f$.

Write $f = f_0 + f_1 t^n + f_2 t^{2n} + \cdots$ and $g = g_0 + g_1 t^n + g_2 t^{2n} + \cdots$. Define $q_1$ and $r_1$ by $1 + q_1 t^n = f_0 g_0$ and $r_1 = f_1 g_0 \mod t^n$. Then $q_1 = -(q_1 + r_1) g_0 \mod t^n$.

Given $f_0, f_1, g_0$, Brent suggested computing $f_0 g_0$, hence $q_1; f_1 g_0$, hence $r_1$; and then $(q_1 + r_1) g_0$, hence $g_1$. Each multiplication can be done with 3 transforms, for a total of 9 transforms to compute $g \mod t^{2n}$ given $g \mod t^n$. The work to compute $g \mod t^n$ by the same method recursively is comparable to $9/2 + 9/4 + \cdots = 9$ transforms.

However, each multiplication uses the same intermediate result $g_0^*$, so only 7 transforms are required: $f_0^*, f_1^*, g_0^*, f_0 g_0, f_1 g_0, (q_1 + r_1)^*$, and $(q_1 + r_1) g_0$.

Higher-order iterations. Define $q_2, q_3, r_2, r_3$ by $(f_0 + f_1 t^n)(g_0 + g_1 t^n) = 1 + q_2 t^n + q_3 t^{3n}$ and $r_2 + r_3 t^n = (f_2 + f_3 t^n)(g_0 + g_1 t^n) \mod t^{2n}$. Then $g_2 + g_3 t^n = -(q_2 + r_2 + (q_3 + r_3) t^n)(g_0 + g_1 t^n) \mod t^{2n}$.

Brent suggested first computing $g_1$ as discussed above, then using size-4n FFTs to multiply $f_0 + f_1 t^n$, $f_2 + f_3 t^n$, and $q_2 + r_2 + (q_3 + r_3) t^n \mod g_0 + g_1 t^n$.

However, it is wasteful to feed $f_0$ and $g_0$ to two different sizes of FFTs. One can do better by sticking with size-2n FFTs: compute $g_1^*, f_1 g_0 + f_0 g_1, f_2 g_0, f_2 g_0, f_2 g_0, g_2 g_0, (q_2 + r_2)^*, (q_3 + r_3)^*, (q_4 + r_4) g_0$, and $(q_2 + r_2) g_0 + (q_3 + r_3) g_0$.

The total is 18 transforms to compute $g \mod t^{4n}$ from $g \mod t^n$. The work to compute $g \mod t^n$ by the same method recursively is comparable to $18/4 + 18/16 + \cdots = 6$ transforms.

The same idea can be applied repeatedly. One can compute $g \mod t^{8n}$ from $g \mod t^n$ with just 40 transforms, for example. The work to compute $g \mod t^n$ recursively is comparable to $5.5 + 1.5/(2^e - 1)$ transforms with an order-2$^e$ iteration. One should select $e$ as a function of $n$ to avoid excessive overhead.

Notes. The idea of using transforms is standard. For example, it is well known that squaring takes only 2 transforms while multiplication takes 3. See [3, section 12] for references.

A 6-transform bound for reciprocals, presumably using an order-4 iteration as above, was announced by Schönhage, Grotefeld, and Vetter in [7, page 213].

3. Quotients

Let $f, g, h$ be power series with $fg = 1$. This section considers the problem of computing $hg$ given $h$ and $f$. 
Brent suggested computing \( g \mod t^n \), as discussed above, and then multiplying \( g \mod t^n \) by \( h \mod t^n \). This takes time comparable to 8.6 transforms if \( g \mod t^n \) is computed with an order-16 iteration. However, one can do better by reusing the intermediate results from the computation of \( g \).

Write \( g = g_0 + g_1 t^n + g_2 t^{2n} + \cdots \). Section 2 shows how to compute \( g_0 \) in time comparable to 5.6 transforms, and then \( g_0^*, g_1^*, g_2^*, g_3^*, g_4^*, g_5^*, g_6^*, g_7^* \) in 40 transforms. Then computing \( g_4^*, g_5^*, g_6^*, g_7^* \) takes just 4 more transforms, and computing \( h g \mod t^{2n} \) takes 16 more transforms.

The upshot is that computing \( h g \mod t^n \) takes time comparable to 8.2 transforms.

**Logarithms.** Let \( f \) be a power series with constant coefficient 1. Then one can compute \( \log f = I(D(f)/f) \) in time comparable to 8.2 transforms.

4. Square roots

Let \( f, g, h \) be power series with \( fg = 1 \) and \( h = f^2 \). This section considers two problems: computing \( f \) and \( g \) given \( h \); and computing \( f \) given \( h \).

Write \( f = f_0 + f_1 t^n + f_2 t^{2n} + \cdots \), \( g = g_0 + g_1 t^n + g_2 t^{2n} + \cdots \), and \( h = h_0 + h_1 t^n + h_2 t^{2n} + \cdots \). Brent suggested (among other techniques) computing \( f \) from the standard Newton iteration: \( f_1 t^n = -(f_0^2 - h)/2f_0 \mod t^{2n} \); \( f_2 t^{2n} + f_3 t^{3n} = -((f_0 + f_1 t^n)^2 - h)/2(f_0 + f_1 t^n) \mod t^{4n} \); etc. However, there is some overlap between each step and the next. For example, one can reuse \( g_0 = 1/f_0 \mod t^n \) in the computation of \( g_0 + g_1 t^n = 1/(f_0 + f_1 t^n) \mod t^{2n} \).

So define \( q_1, r_1 \) as in section 2. Define \( s_1 \) by \( s_1 t^n = f_0^2 - h_0 - h_1 t^n \); \( s_2 \) and \( s_3 \) by \( s_2 t^{2n} + s_3 t^{3n} = (f_0 + f_1 t^n)^2 - h_0 - h_1 t^n - h_2 t^{2n} + h_3 t^{3n} \); and so on. Then

\[
f_1 = (-1/2)s_1 g_0 \mod t^n, \quad f_2 + f_3 t^n = (-1/2)(s_2 + s_3 t^n)(g_0 + g_1 t^n) \mod t^{2n},
\]

Starting from \( f_0, g_0, h_0, h_1 \), one can compute \( f_1 \) with 5 transforms: \( f_0^*, f_0^2, f_1^*, g_0^*, s_1 g_0 \). One can then compute \( g_1 \) with 5 transforms: \( f_1^*, f_0 g_0, f_1 g_0, (q_1 + r_1)^*, (q_1 + r_1)g_0 \). One can then compute \( f_2, f_3 \) with 7 transforms; then \( g_2, g_3 \) with 10 transforms; and so on.

The work to compute both \( f \mod t^n \) and \( g \mod t^n \) recursively is comparable to 9 transforms with an order-4 iteration. The work to compute \( f \mod t^n \) is comparable to 6.5 transforms.

**Notes.** Bailey in [1] reported a square root time comparable to 21 transforms.

5. Exponentials

Let \( f, g, h \) be power series with \( fg = 1 \) and \( f = \exp h \). This section considers the problem of computing \( f \) and \( g \) given \( h \).

Write \( f = f_0 + f_1 t^n + f_2 t^{2n} + \cdots \), \( g = g_0 + g_1 t^n + g_2 t^{2n} + \cdots \), \( h = h_0 + h_1 t^n + h_2 t^{2n} + \cdots \), \( D(f) = a_0 + a_1 t^n + a_2 t^{2n} + \cdots \), and \( D(h) = b_0 + b_1 t^n + b_2 t^{2n} + \cdots \). Define \( q_1, r_1 \) as in section 2. Define \( s_1 \) by \( s_1 t^n = h - I(a_0 g_0 - b_0 q_1 t^n) \mod t^{2n} \); define \( s_2, s_3 \) by

\[
s_2 t^{2n} + s_3 t^{3n} = h - I((a_0 + a_1 t^n)(g_0 + g_1 t^n) - (b_0 + b_1 t^n)(q_2 t^{2n} + q_3 t^{3n})) \mod t^{4n},
\]

and so on. Then \( f_1 = f_0 s_1 \mod t^n \); \( f_2 + f_3 t^n = (f_0 + f_1 t^n)(s_2 + s_3 t^n) \mod t^{2n} \); etc. XXX still have to check these run times.

Starting from \( f_0, g_0, h_0, h_1 \), one can compute \( f_1 \) with 10 transforms: \( f_0^*, f_0^2, f_0 g_0, a_0^*, a_0 g_0, b_0^*, b_0 q_1, s_1^*, f_0 s_1 \). One can then compute \( g_1 \) with 4 transforms: \( f_1^* \),
\( f_1 g_0, (q_1 + r_1)^*, (q_1 + r_1) g_0 \). One can then compute \( f_2, f_3 \) with 15 transforms; then \( g_2, g_3 \) with 8 transforms; and so on.

The work to compute both \( f \mod t^n \) and \( g \mod t^n \) recursively is comparable to \( 37/3 \) transforms with an order-4 iteration. The work to compute \( f \mod t^n \) is comparable to \( 31/3 \) transforms.

6. Implementation results

XXX This is a first draft! But presumably the timings will be in line with the theoretical estimates shown below.

recip: (9, 6) 5.6
quo or log: (12, 9) 8.2
sqrt: (16, 5) 6.5
sqrt and isqrt: (16, 5) 9
exp: (22) 31/3

I also still have to analyze “natural” order-3 and order-4 iterations.

References