Aimed at Math. Comp. I need to do some software verification first.

REMOVING REDUNDANCY IN HIGH-PRECISION NEWTON ITERATION

DANIEL J. BERNSTEIN

Abstract. This paper presents new algorithms for several high-precision operations in the power series ring $\mathbf{C}[[x]]$. Compared to computing n coefficients of a product in $\mathbf{C}[[x]]$, computing n coefficients of a reciprocal in $\mathbf{C}[[x]]$ takes $1.5+o(1)$ times longer; a quotient or logarithm, $2.16666\ldots+o(1)$ times longer; a square root, $1.83333\ldots+o(1)$ times longer; an exponential, $2.83333\ldots+o(1)$ times longer. Previous algorithms had worse constants. The same ideas apply to high-precision computations in $\mathbf{R}$, $\mathbf{Q}_p$, etc.

1. Introduction

Let $f \in \mathbf{C}[[x]]$ be a power series with constant coefficient 1. How can one compute $1/f$?

The standard answer is Newton's method, which computes $(1/f) \bmod x^{2n}$ from $(1/f) \bmod x^n$ with a few size-n polynomial multiplications: $(1/f) \bmod x^{2n} = g_0 + (1 - fg_0)g_0 \bmod x^{2n}$ where $g_0 = (1/f) \bmod x^n$. One can compute g_0 by the same method recursively. One can use FFTs to carry out the polynomial multiplications using $O(n \lg n)$ operations in $\mathbf{R}$.

Similar comments apply to $f^{1/2}$, $\log f$, et al.

The point of this paper is that there are certain redundancies inside typical FFT-based Newton iterations: known coefficients of products (Section 3), repeated multiplicands (Section 4), sums of products (also Section 4), and something for which I don't have a catchy name (Section 5). This paper systematically eliminates the redundancies to achieve various constant-factor speedups.

The bottom line is that one can compute

- $(1/f) \bmod x^n$ using $(36 + o(1))n \lg n$ operations in $\mathbf{R}$;
- $(g/f) \bmod x^n$ using $(52 + o(1))n \lg n$ operations in $\mathbf{R}$;
- $((1/f) \bmod x^n, (g/f) \bmod x^n)$ using $(56 + o(1))n \lg n$ operations in $\mathbf{R}$;
- $f^{1/2} \bmod x^n$ using $(44 + o(1))n \lg n$ operations in $\mathbf{R}$;
- $(f^{1/2} \bmod x^n, f^{-1/2} \bmod x^n)$ using $(60 + o(1))n \lg n$ operations in $\mathbf{R}$;
- $\log f \bmod x^n$ using $(52 + o(1))n \lg n$ operations in $\mathbf{R}$;
- $\exp f \bmod x^n$, when f has constant coefficient 0, using $(68 + o(1))n \lg n$ operations in $\mathbf{R}$; and
- $(\exp f \bmod x^n, (1/\exp f) \bmod x^n)$ using $(84 + o(1))n \lg n$ operations in $\mathbf{R}$.

For comparison: Computing $fg \bmod x^n$ uses $(24 + o(1))n \lg n$ operations in $\mathbf{R}$.

Date: 2004.03.09. Permanent ID of this document: `def7f1e35fb654671c6f767b16b93d50`.
2000 *Mathematics Subject Classification.* Primary 68W30; Secondary 65Y20.
The author was supported by the National Science Foundation under grants DMS–9600083, DMS–9970409, and DMS–0140542.

Generalizations. Newton's method is not limited to $\mathbf{C}[[x]]$. For example, the same formula $g_0 + (1 - fg_0)g_0$ can be used

- to compute high-precision reciprocals of power series f in $k[[x]]$ using very few operations in k, where k is a finite field;
- to compute high-precision reciprocals of real numbers f using very few bit operations;
- to compute high-precision reciprocals of p-adic numbers f using very few bit operations;

and so on. The details of the algorithms for p-adic numbers rely on some easy roundoff error analysis; see, e.g., [2, section 21]. The details of the algorithms for real numbers rely on some not-so-easy roundoff error analysis; see, e.g., [2, section 8]. Some functions, such as $f \mapsto \log f$, require completely different methods for real numbers; see, e.g., [5], [3, Sections 15–16], and [4].

In each setting, multiplication can be performed in essentially linear time; see, e.g., [3, Sections 2–5]. Newton's method then takes essentially linear time.

The techniques of this paper can be adapted to each setting. For example, the same constant-factor speedups apply to division in each setting. I have focused on $\mathbf{C}[[x]]$ in this paper because it is the simplest case.

Previous work. Shortly after it became widely known that one could multiply integers in essentially linear time, Cook in [6, pages 77–86] presented an essentially-linear-time algorithm for high-precision reciprocals in $\mathbf{R}$. Cook also mentioned, without giving details, that Newton's method could be used to compute square roots and other algebraic functions.

Sieveking in [11] published an essentially-linear-time algorithm for reciprocals in $\mathbf{C}[[x]]$. Kung in [9] pointed out that Sieveking's algorithm was an example of Newton's method and was analogous to Cook's algorithm. Brent in [5] stated essentially-linear-time algorithms to compute square roots, logarithms, et al. in $\mathbf{R}$ and in $\mathbf{C}[[x]]$.

There have been several attempts to squeeze constant factors out of the time taken to compute reciprocals. The simplest Newton iteration for reciprocals is $4 + o(1)$ times slower than multiplication; see, e.g., [1, Section 4]. Brent in [5] presented an algorithm achieving $3 + o(1)$. Schönhage, Grotefeld, and Vetter in [10, page 256] announced $2 + o(1)$ without giving details. This paper presents a messy $1.5 + o(1)$ algorithm (which I discovered in 1999 and announced in 2000) and a clean $1.66666\ldots + o(1)$ algorithm (2002).

For quotients (and thus logarithms in the $\mathbf{C}[[x]]$ case), Brent's algorithm achieved $4+o(1)$. Schönhage, Grotefeld, and Vetter announced $3+o(1)$ without giving details. Karp and Markstein in [7] presented an idea for saving time but did not analyze its speed. In 1998, after I published an early draft of this paper, Rob Harley told me that he had achieved $2.625 + o(1)$ a few years earlier. This paper presents a messy $2.16666\ldots + o(1)$ algorithm (1999) and a clean $2.33333\ldots + o(1)$ algorithm (2002).

For square roots, the simplest Newton iteration achieves $7 + o(1)$; see, e.g., [1, Section 4]. Brent's algorithm achieved $4.5 + o(1)$. Paul Zimmermann achieved $3.5 + o(1)$ in unpublished work. This paper presents a clean $1.83333\ldots + o(1)$ algorithm (2004; same bound achieved messily in 1999).

For exponentials (in the $\mathbf{C}[[x]]$ case), Brent's algorithm achieved $7.33333\ldots + o(1)$. This paper presents a messy $2.83333\ldots + o(1)$ algorithm (1999) and a clean $3.33333\ldots + o(1)$ algorithm (2004).

2. Newton's method

This section applies Newton's method to reciprocals, quotients, square roots, logarithms, and exponentials. The resulting algorithms have various redundancies that will be removed in subsequent sections. The same ideas can also be applied to other functions computed with Newton's method.

Model of computation. A polynomial $f_0 + f_1x + f_2x^2 + \cdots \in \mathbf{C}[x]$, known to be of degree below n, is represented as a sequence of n coefficients $(f_0, f_1, \ldots, f_{n-1})$. Each complex number c is represented as a pair $(\operatorname{re} c, \operatorname{im} c)$ of real numbers. Thus each polynomial is represented as a sequence of real numbers.

A power series $f = f_0 + f_1x + f_2x^2 + \cdots \in \mathbf{C}[[x]]$ is represented as a polynomial $f \bmod x^n = f_0 + f_1x + \cdots + f_{n-1}x^{n-1}$ approximating f to a specified precision n.

Operations on polynomials and power series thus boil down to operations on real numbers. The only operations in $\mathbf{R}$ used in this paper are additions, subtractions, and multiplications; I evaluate algorithm speed by counting these operations. For example, a complex multiplication takes 6 operations in $\mathbf{R}$.

Tools: polynomial multiplication. Given two polynomials $u, v \in \mathbf{C}[x]$ with $\deg u < n$ and $\deg v < n$, one can compute uv using $(24 + o(1))n \lg n$ operations in $\mathbf{R}$. This is the bottleneck in all the algorithms in this section. See [3, Section 2] for the underlying FFT algorithms and [3, Section 4] for multiplication via the FFT.

In particular, say $f, g \in \mathbf{C}[[x]]$. Given $f \bmod x^n$ and $g \bmod x^n$, one can compute $fg \bmod x^n = (f \bmod x^n)(g \bmod x^n) \bmod x^n$ using $(24 + o(1))n \lg n$ operations in $\mathbf{R}$.

There are two different methods to achieve the constant 24: the "split-radix FFT" and the "real-factor FFT." Older articles on Newton's method sometimes use slower FFT algorithms. Perhaps faster FFT algorithms will be discovered someday. One should, of course, substitute the best available FFTs into the algorithms in this paper.

Reciprocal. Let $f \in \mathbf{C}[[x]]$ be a power series with constant coefficient 1, as in Section 1. Define $g = 1/f$. Consider the problem of computing a high-precision approximation to g given a high-precision approximation to f.

If $g \bmod x^n = g_0$ then $g \bmod x^{2n} = g_0 - (fg_0 - 1)g_0 \bmod x^{2n} = g_0 - \epsilon g_0 \bmod x^{2n}$ where $\epsilon = (fg_0 - 1) \bmod x^{2n}$. The same formula holds with $2n - 1$ in place of $2n$; but for ease of exposition I will focus on $2n$ and ignore $2n - 1$.

Multiplying $f \bmod x^{2n}$ by g_0 uses $(48 + o(1))n \lg n$ operations in $\mathbf{R}$. Subtracting 1 is fast. Reducing modulo x^{2n} to obtain $\epsilon = ((f \bmod x^{2n})g_0 - 1) \bmod x^{2n}$ is fast: this is simply throwing away coefficients. Multiplying ϵ by g_0 uses $(48 + o(1))n \lg n$ operations in $\mathbf{R}$. Reducing modulo x^{2n} is fast. Subtracting from g_0 to obtain $g \bmod x^{2n}$ is fast.

The total effort is $(96+o(1))n \lg n$ operations in $\mathbf{R}$ to compute $g \bmod x^{2n}$ starting from $g \bmod x^n$. Therefore one can recursively compute $g \bmod x^n$ for any desired n, starting from $g \bmod x^1 = 1$, using $(96 + o(1))n \lg n$ operations in $\mathbf{R}$.

Quotient. Let $f \in \mathbf{C}[[x]]$ be a power series with constant coefficient 1. Let $h \in \mathbf{C}[[x]]$ be a power series. Define $g = 1/f$ and $q = hg = h/f$. Consider the problem of computing a high-precision approximation to q given high-precision approximations to f and h.

The obvious strategy to compute a quotient is as a reciprocal followed by a product: first compute $g = 1/f$, then compute $q = hg$. This strategy takes $(120 + o(1))n \lg n$ operations in $\mathbf{R}$.

Karp and Markstein in [7] suggested a different strategy. Recall the algorithm for computing $g \bmod x^{2n}$ from $g_0 = g \bmod x^n$: one has $g \bmod x^{2n} = g_0(1-\epsilon) \bmod x^{2n}$ where $\epsilon = (fg_0 - 1) \bmod x^{2n}$. One can multiply h by $1 - \epsilon$ and then by g_0, without computing $g_0(1-\epsilon)$. This strategy also takes $(120 + o(1))n \lg n$ operations in $\mathbf{R}$.

Subsequent sections of this paper will accelerate the second strategy slightly more than the first strategy. On the other hand, the second strategy produces solely q, while the first strategy also produces g, which is useful for further divisions by f. The second strategy also has the temporary disadvantage of being patented.

Square root. Let $h \in \mathbf{C}[[x]]$ be a power series with constant coefficient 1. Define $f = h^{1/2}$ and $g = 1/f = h^{-1/2}$. Consider the problem of computing high-precision approximations to f and g given a high-precision approximation to h.

The reader may be interested solely in $h^{1/2}$ and may wonder why $h^{-1/2}$ is being considered at the same time. Answer: Newton's method for the equation $f^2 - h = 0$ says that if f_0 is a good approximation to f then $f_0 - (f_0^2 - h)/2f_0$ is a better approximation. The reciprocal $1/f_0$ is approximately $h^{-1/2}$.

Write $f_0 = f \bmod x^n$, $g_0 = g \bmod x^n$, and $\delta = f_0^2 - h \bmod x^{2n}$. Then $f \bmod x^{2n} = f_0 - \delta/2f_0 \bmod x^{2n} = f_0 - \delta g_0/2 \bmod x^{2n}$.

Squaring f_0 takes $(24 + o(1))n \lg n$ operations in $\mathbf{R}$. Subtracting $h \bmod x^{2n}$ to obtain δ is fast. Multiplying δ by g_0 takes $(48 + o(1))n \lg n$ operations in $\mathbf{R}$. Reducing modulo x^{2n} is fast. Multiplying by 1/2 and subtracting from f_0, to obtain $f \bmod x^{2n}$, is fast. Computing $g \bmod x^{2n}$ then takes $(96 + o(1))n \lg n$ operations in $\mathbf{R}$.

The total effort is $(168 + o(1))n \lg n$ operations in $\mathbf{R}$ to compute both $f \bmod x^{2n}$ and $g \bmod x^{2n}$ starting from $f \bmod x^n$ and $g \bmod x^n$. Therefore one can recursively compute $f \bmod x^n$ and $g \bmod x^n$, starting from $f \bmod x^1 = g \bmod x^1 = 1$, using $(168 + o(1))n \lg n$ operations in $\mathbf{R}$.

If g is not desired then one can skip the computation of g in the last step of the recursion. One obtains $f \bmod x^n$ using $(120 + o(1))n \lg n$ operations in $\mathbf{R}$.

Logarithm. Let $f \in \mathbf{C}[[x]]$ be a power series with constant coefficient 1. Define $g = \log f$. Consider the problem of computing a high-precision approximation to g given a high-precision approximation to f.

Brent in [5, Section 13] suggested computing g from the formula $D(g) = D(f)/f$, where $D(\sum a_i x^i) = \sum i a_i x^i$. Starting from $f \bmod x^n$ one multiplies the ith coefficient by i to compute $D(f) \bmod x^n$; then divides by f, as above, to compute $D(g) \bmod x^n$; then multiplies the ith coefficient by $1/i$ (for $i \geq 1$) to compute g.

The bottleneck here is the quotient, which uses $(120 + o(1))n \lg n$ operations in $\mathbf{R}$.

Exponential. Let $h \in \mathbf{C}[[x]]$ be a power series with constant coefficient 0. Define $f = \exp h$ and $g = 1/f$. Consider the problem of computing high-precision approximations to f and g, given a high-precision approximation to h.

Write $f_0 = f \bmod x^n$, $g_0 = g \bmod x^n$, $\delta = f_0 g_0 - 1$, and $\epsilon = (\log f_0) - h \bmod x^{2n}$. Then $f \bmod x^{2n} = f_0 - \epsilon f_0 \bmod x^{2n}$ and $D(\epsilon) \equiv (1+\delta)D(\epsilon) \equiv (1+\delta)D(f_0)/f_0 - (1+\delta)D(h) \equiv D(f_0)g_0 - D(h) - \delta D(h_0) \pmod{x^{2n}}$.

Multiplying f_0 by g_0 takes $(24 + o(1))n \lg n$ operations in $\mathbf{R}$. Subtracting 1, to obtain δ, is fast. Multiplying $D(f_0)$ by g_0 takes $(24 + o(1))n \lg n$ operations in $\mathbf{R}$. Multiplying δ by $D(h_0)$ takes $(48 + o(1))n \lg n$ operations in $\mathbf{R}$. Subtracting to obtain $D(f_0)g_0 - \delta D(h_0)$ is fast. Reducing modulo x^{2n} is fast. Subtracting $D(h) \bmod x^{2n}$, to obtain $D(\epsilon)$, is fast. Multiplying ϵ by f_0 takes $(48 + o(1))n \lg n$ operations in $\mathbf{R}$. Subtracting from f_0, and reducing modulo x^{2n} to obtain $f \bmod x^{2n}$, is fast. Computing $g \bmod x^{2n}$ then takes $(96 + o(1))n \lg n$ operations in $\mathbf{R}$.

The total effort is $(240 + o(1))n \lg n$ operations in $\mathbf{R}$ to compute both $f \bmod x^{2n}$ and $g \bmod x^{2n}$ starting from $f \bmod x^n$ and $g \bmod x^n$. Therefore one can recursively compute $f \bmod x^n$ and $g \bmod x^n$, starting from $f \bmod x^1 = g \bmod x^1 = 1$, using $(240 + o(1))n \lg n$ operations in $\mathbf{R}$.

If g is not desired then, as before, one can skip the computation of g in the last step of the recursion. One obtains $f \bmod x^n$ using $(192 + o(1))n \lg n$ operations in $\mathbf{R}$.

3. Known coefficients of products

Here is a common form of redundancy: an algorithm computes a polynomial product uv when some top or bottom coefficients of uv are already known in other ways. This section identifies and eliminates this form of redundancy in each of the algorithms of Section 2.

The same form of redundancy also shows up outside the context of Newton's method. I should add references here.

Tools: polynomial multiplication modulo $x^{2n} - 1$. Given two polynomials $u, v \in \mathbf{C}[x]$ with $\deg u < 2n$ and $\deg v < 2n$, one can compute $uv \bmod x^{2n} - 1$ using $(24 + o(1))n \lg n$ operations in $\mathbf{R}$, provided that n is ultrasmooth.

Here **ultrasmooth** means that $n = 2^k m$ for an odd integer $m \geq 1$ and an integer $k \geq m^2 - 1$. For example, the integers $2^{10000}50$, $2^{10000}51$, $2^{10000}52$, $\ldots$, $2^{10000}99$, $2^{10001}50$, $\ldots$ are ultrasmooth. Observe that the ratios of successive ultrasmooth integers converge to 1.

The algorithm to compute $uv \bmod x^{2n} - 1$ is the same FFT-based algorithm that was used in Section 2 to compute uv when $\deg u < n$ and $\deg v < n$. This section exploits the fact that the algorithm actually allows larger inputs u, v, and multiplies them modulo $x^{2n} - 1$.

(For readers familiar with FFTs: Say $n = 2^k m$ where m is odd. One can compute a size-$2n$ FFT by combining a split-radix FFT with a brute-force size-m DFT. This takes $8nk + O(mn)$ operations in $\mathbf{R}$, and therefore $(8 + o(1))n \lg n$ operations in $\mathbf{R}$ if n is ultrasmooth.)

The restriction to ultrasmooth integers n does not pose a problem. If n is not ultrasmooth then one can find the smallest ultrasmooth integer $n' \geq n$ and multiply modulo $x^{2n'} - 1$. This uses $(24 + o(1))n' \lg n'$ operations in $\mathbf{R}$, which is the same as $(24 + o(1))n \lg n$ operations in $\mathbf{R}$ since $n' \in (1 + o(1))n$.

There are two ways to achieve further $1 + o(1)$ speedups when n is not a power of 2. First, one can reduce n'/n by expanding the set of ultrasmooth integers; this requires more work in the multiplication algorithm. Second, one can allow moduli other than $x^{2n} - 1$; this allows some slightly faster multiplication algorithms. The important property of the moduli $x^{2n} - 1$, as the reader will see, is that one can quickly divide by x^n modulo $x^{2n} - 1$.

How to exploit known coefficients. The point of this section is that, if uv is known except for a stretch of $2n$ unknown coefficients, then one can compute uv by computing $uv \bmod x^{2n} - 1$.

Specifically, say one knows a polynomial w such that $\deg(uv-w) < x^{3n}$ and $uv \equiv w \pmod{x^n}$. I claim that $(uv-w)/x^n$ is the same as $(x^n(uv-w)) \bmod x^{2n}-1$: both sides are polynomials of degree below $2n$, and they are congruent modulo $x^{2n} - 1$. Hence $uv - w = x^n((x^n(uv - w)) \bmod x^{2n} - 1)$. By computing $uv \bmod x^{2n} - 1$, subtracting $w \bmod x^{2n} - 1$, multiplying by x^n modulo $x^{2n} - 1$ (which boils down to swapping top and bottom coefficients), and multiplying the result by x^n, one obtains $uv - w$.

Similarly, if one knows a polynomial w such that $\deg(uv - w) < x^{2n}$ and $uv \equiv w \pmod{x^n}$, then one can compute $uv - w$ by computing $uv \bmod x^n - 1$.

Reciprocal. Consider again the problem of computing $g = 1/f$ given f. Write $g_0 = g \bmod x^n$ and $\epsilon = ((f \bmod x^{2n})g_0 - 1) \bmod x^{2n}$; recall that $g \bmod x^{2n} = g_0 - \epsilon g_0 \bmod x^{2n}$.

The algorithm from Section 2 computed $(f \bmod x^{2n})g_0$ as a generic product of degree below $4n$. But the top n coefficients of the product are known: $\deg g_0 < n$ so $\deg((f \bmod x^{2n})g_0) < 3n$. The bottom n coefficients of the product are also known: $((f \bmod x^{2n})g_0) \bmod x^n = fg \bmod x^n = 1$. Hence it suffices to compute $(f \bmod x^{2n})g_0 \bmod x^{2n} - 1$.

Similarly, computing ϵg_0 boils down to computing $\epsilon g_0 \bmod x^{2n} - 1$.

The total effort drops by a factor of $2 + o(1)$, because the multiplications are now half-size. This algorithm takes $(48 + o(1))n \lg n$ operations in $\mathbf{R}$ to compute $g \bmod x^{2n}$ from $g \bmod x^n$, or to compute $g \bmod x^n$ from scratch recursively.

Quotient or logarithm. Consider again the problem of computing $q = h/f$ given f and h. Define g, g_0, ϵ as above.

The second strategy from Section 2 multiplied h by $1 - \epsilon$ modulo x^{2n}, and then multiplied the result by g_0 modulo x^{2n}. Notice that $h(1 - \epsilon) = h - h\epsilon \equiv h - (h \bmod x^n)\epsilon \pmod{x^{2n}}$. The product $(h \bmod x^n)\epsilon$ can be computed modulo $x^{2n} - 1$: it has degree below $3n$ and its bottom n coefficients are known to be 0. The final multiplication by g_0 can be performed modulo $x^{3n} - 1$: the product has degree below $3n$.

The initial computation of g_0 takes $(48 + o(1))n \lg n$ operations in $\mathbf{R}$. The computation of ϵ takes $(24 + o(1))n \lg n$ operations in $\mathbf{R}$. The multiplication by h takes $(24 + o(1))n \lg n$ operations in $\mathbf{R}$. The final multiplication by g_0 takes $(36 + o(1))n \lg n$ operations in $\mathbf{R}$. The total effort is $(132 + o(1))n \lg n$ operations in $\mathbf{R}$ to compute $q \bmod x^{2n}$; i.e., $(66 + o(1))n \lg n$ operations in $\mathbf{R}$ to compute $q \bmod x^n$.

Square root. Consider again the problem of computing $f = h^{1/2}$ and $g = h^{-1/2}$ given h. Write $f_0 = f \bmod x^n$, $g_0 = g \bmod x^n$, and $\delta = f_0^2 - h \bmod x^{2n}$. Recall that $f \bmod x^{2n} = f_0 - (1/2)\delta g_0 \bmod x^{2n}$.

The square f_0^2 can be computed modulo $x^n - 1$: it has degree below $2n$, and its bottom n coefficients are known to match $h \bmod x^n$. Similarly, the product δg_0 can be computed modulo $x^{2n} - 1$. The total effort is $(36 + o(1))n \lg n$ operations in $\mathbf{R}$

to compute $f \bmod x^{2n}$ from $f \bmod x^n$ and $g \bmod x^n$, and hence $(84 + o(1))n \lg n$ operations in $\mathbf{R}$ to compute both $f \bmod x^{2n}$ and $g \bmod x^{2n}$ from $f \bmod x^n$ and $g \bmod x^n$.

Computing $f \bmod x^n$ and $g \bmod x^n$ from scratch takes $(84 + o(1))n \lg n$ operations in $\mathbf{R}$. Computing just $f \bmod x^n$ takes $(60 + o(1))n \lg n$ operations in $\mathbf{R}$. As in the case of reciprocals, the total effort has dropped by a factor $2 + o(1)$, because each multiplication is half the size it was before.

Exponential. Consider again the problem of computing $f = \exp h$ and $g = 1/f$, given h.

Write $f_0 = f \bmod x^n$, $g_0 = g \bmod x^n$, $h_0 = h \bmod x^n$, $\delta = f_0 g_0 - 1$, and $\epsilon = (\log f_0) - h \bmod x^{2n}$. Recall that $f \bmod x^{2n} = f_0 - \epsilon f_0 \bmod x^{2n}$ and $D(\epsilon) = D(f_0)g_0 - D(h) - \delta D(h_0) \bmod x^{2n}$.

The product $f_0 g_0$ can be computed modulo $x^n - 1$: it has degree below $2n$, and it is congruent to 1 modulo x^n.

The product $D(f_0)g_0$ can be computed modulo $x^n - 1$: it has degree below $2n$, and it is congruent to $D(h_0)$ modulo x^n.

The product $\delta D(h_0)$ can be computed modulo $x^{2n} - 1$: it has degree below $3n$, and it is congruent to 0 modulo x^n.

The product ϵf_0 can be computed modulo $x^{2n} - 1$: it has degree below $3n$, and it is congruent to 0 modulo x^n.

The total effort is $(72 + o(1))n \lg n$ operations in $\mathbf{R}$ to compute $f \bmod x^{2n}$ from $f \bmod x^n$ and $g \bmod x^n$. and hence $(120 + o(1))n \lg n$ operations in $\mathbf{R}$ to compute both $f \bmod x^{2n}$ and $g \bmod x^{2n}$ from $f \bmod x^n$ and $g \bmod x^n$. Computing $f \bmod x^n$ and $g \bmod x^n$ from scratch recursively takes $(120 + o(1))n \lg n$ operations in $\mathbf{R}$. Computing just $f \bmod x^n$ takes $(96 + o(1))n \lg n$ operations in $\mathbf{R}$. The total effort has once again dropped by a factor $2 + o(1)$.

4. Repeated multiplicands and sums of products

Here are two common forms of redundancy. First, an algorithm computes two separate products uv and uw that share an input u. Second, an algorithm computes a sum of separate products, such as $tu + vw$.

This section identifies and eliminates these forms of redundancy in each of the algorithms of Section 3.

The same forms of redundancy also show up outside the context of Newton's method. I should add references here.

Tools: the FFT. This section relies on further knowledge of the structure of FFT-based multiplication.

The **size-$2n$ FFT** is a function F_{2n} from $\mathbf{C}[x]/(x^{2n} - 1)$ to $\mathbf{C}^{2n}$. This function is a $\mathbf{C}$-algebra morphism. In particular, it preserves addition, subtraction, and multiplication: $F_{2n}(u \pm v) = F_{2n}(u) \pm F_{2n}(v)$, and $F_{2n}(uv) = F_{2n}(u)F_{2n}(v)$. Here the addition, subtraction, and multiplication in $\mathbf{C}^{2n}$ are componentwise operations that take $O(n)$ operations in $\mathbf{R}$.

Computing $F_{2n}(u)$, given $u \bmod x^{2n} - 1$, takes $(8 + o(1))n \lg n$ operations in $\mathbf{R}$ when n is ultrasmooth. Computing $u \bmod x^{2n} - 1$, given $F_{2n}(u)$, also takes $(8 + o(1))n \lg n$ operations in $\mathbf{R}$ when n is ultrasmooth.

Furthermore, given $F_{2n}(u)$ and $u \bmod x^{2n} + 1$, one can compute $F_{4n}(u)$ with $(8 + o(1))n \lg n$ operations in $\mathbf{R}$.

Section 3's procedure for multiplication modulo $x^{2n}-1$ is implemented as follows. Starting from u, v, one computes $F_{2n}(u)$ and $F_{2n}(v)$; then multiplies in $\mathbf{C}^{2n}$ to obtain $F_{2n}(uv)$; then inverts F_{2n} to obtain $uv \bmod x^{2n}-1$.

How to exploit repeated multiplicands at the same size: FFT caching. There is no need to compute $F_{2n}(u)$ more than once.

For example, to compute $u^2 \bmod x^{2n}-1$, one can compute $F_{2n}(u)$, then square in $\mathbf{C}^{2n}$, then invert F_{2n}. This takes only $(16+o(1))n\lg n$ operations in $\mathbf{R}$, rather than $(24+o(1))n\lg n$ operations in $\mathbf{R}$.

Similarly, computing both $uv \bmod x^{2n}-1$ and $uw \bmod x^{2n}-1$ takes only $(40+o(1))n\lg n$ operations in $\mathbf{R}$. The intermediate result $F_{2n}(u)$ from the computation of uv is saved and reused in the computation of uw.

How to exploit repeated multiplicands at double size: FFT doubling. Rather than separately computing $F_{2n}(u)$ and $F_{4n}(u)$, one can compute $F_{2n}(u)$ and then use it along with $u \bmod x^{2n}+1$ to compute $F_{4n}(u)$. I learned this speedup from a recent preprint by Robert Kramer.

For example, in computing $uv \bmod x^{2n}-1$ and $uw \bmod x^{4n}-1$, one computes both $F_{2n}(u)$ and $F_{4n}(u)$. Half of the time in computing $F_{4n}(u)$ can be eliminated.

How to exploit sums of products: FFT addition. Rather than inverting F_{2n} on several inputs and adding the results, one can add the results and then invert F_{2n} once.

For example, to compute $tu+vw \bmod x^{2n}-1$, one can compute $F_{2n}(t)F_{2n}(u)+F_{2n}(v)F_{2n}(w)$ in $\mathbf{C}^{2n}$, and then invert F_{2n}. This takes only $(40+o(1))n\lg n$ operations in $\mathbf{R}$, rather than $(48+o(1))n\lg n$ operations in $\mathbf{R}$,

Reciprocal. Consider again the problem of computing $g = 1/f$ given f. Write $g_0 = g \bmod x^n$. Recall that the algorithm from Section 3 multiplies $f \bmod x^{2n}$ by g_0 modulo $x^{2n}-1$, and then multiplies a certain polynomial ϵ by g_0 modulo $x^{2n}-1$.

The intermediate result $F_{2n}(g_0)$ can be saved and reused, eliminating $(8+o(1))n\lg n$ operations in $\mathbf{R}$. The total effort is now $(40+o(1))n\lg n$ operations in $\mathbf{R}$ to compute $g \bmod x^{2n}$ from $g \bmod x^n$, or to compute $g \bmod x^n$ from scratch recursively.

Quotient or logarithm. Consider again the problem of computing $q = h/f$ given f and h.

Write $g = 1/f$. The first strategy to compute $q \bmod x^{2n}$ was to compute $g \bmod x^{2n}$ and then multiply by $h \bmod x^{2n}$. The multiplication by $h \bmod x^{2n}$ can take advantage of the previous transform of $g \bmod x^n$.

Define g_0, g_1, h_0, h_1 by $g_0 = g \bmod x^n$, $g_0 + g_1x^n = g \bmod x^{2n}$, $h_0 = h \bmod x^n$, and $h_0 + h_1x^n = h \bmod x^{2n}$. Then $q \bmod x^{2n} = g_0h_0 + (g_0h_1 + g_1h_0)x^n \bmod x^{2n}$.

Computing g_0 from scratch takes $(40+o(1))n\lg n$ operations in $\mathbf{R}$. Computing g_1—with $F_{2n}(g_0)$ as an intermediate result—takes another $(40+o(1))n\lg n$ operations in $\mathbf{R}$. Computing $F_{2n}(g_1)$, $F_{2n}(h_0)$, and $F_{2n}(h_1)$ takes another $(24+o(1))n\lg n$ operations in $\mathbf{R}$. Computing $F_{2n}(g_0h_0)$ and $F_{2n}(g_0h_1 + g_1h_0)$ takes $o(1)n\lg n$ operations in $\mathbf{R}$. Computing g_0h_0 and $g_0h_1 + g_1h_0$, and thus $q \bmod x^{2n}$, takes $(16+o(1))n\lg n$ operations in $\mathbf{R}$.

The total effort is $(120+o(1))n\lg n$ operations in $\mathbf{R}$ to compute $g \bmod x^{2n}$ and $q \bmod x^{2n}$; i.e., $(60+o(1))n\lg n$ operations in $\mathbf{R}$ to compute $g \bmod x^n$ and

$q \bmod x^n$. Note that $F_{2n}(g_0)$ and $F_{2n}(g_1)$ are now both cached, so dividing another power series by f to precision n takes only $(16 + o(1))n \lg n$ operations.

The second strategy was to multiply h by $1 - \epsilon_1 x^n$ and then by g_0, where ϵ_1 was a particular polynomial of degree below n. Both g_0 and h_0 are repeated here.

Define $u_1 = h_1 - \epsilon_1 h_0 \bmod x^n$. Then $h(1 - \epsilon_1 x^n) \equiv h_0 + u_1 x^n \pmod{x^{2n}}$, so $q \equiv h(1 - \epsilon_1 x^n) g_0 \equiv h_0 g_0 + u_1 g_0 x^n \pmod{x^{2n}}$.

Computing g_0 from scratch takes $(40 + o(1))n \lg n$ operations in $\mathbf{R}$. Computing $F_{2n}(g_0)$ and ϵ_1 takes $(24 + o(1))n \lg n$ operations in $\mathbf{R}$. Computing $F_{2n}(h_0)$ and $\epsilon_1 h_0$, hence u_1, takes $(24 + o(1))n \lg n$ operations in $\mathbf{R}$. Computing $h_0 g_0$ takes $(8 + o(1))n \lg n$ operations in $\mathbf{R}$. Computing $u_1 g_0$ takes $(16 + o(1))n \lg n$ operations in $\mathbf{R}$.

The total effort is $(112 + o(1))n \lg n$ operations in $\mathbf{R}$ to compute $q \bmod x^{2n}$; i.e., $(56 + o(1))n \lg n$ operations in $\mathbf{R}$ to compute $q \bmod x^n$.

Square root. Consider again the problem of computing $f = h^{1/2}$ and $g = h^{-1/2}$ given h. Write $f_0 = f \bmod x^n$, $g_0 = g \bmod x^n$, and $\delta = f_0^2 - h \bmod x^{2n}$. Recall that $f \bmod x^{2n} = f_0 - (1/2)\delta g_0 \bmod x^{2n}$.

Assume that f_0, g_0, and $F_n(f_0)$ are already known. Computing $f_0^2 \bmod x^n - 1$, hence f_0^2, hence δ, takes $(4 + o(1))n \lg n$ operations in $\mathbf{R}$. Computing $F_{2n}(g_0)$ takes $(8 + o(1))n \lg n$ operations in $\mathbf{R}$. Computing $F_{2n}(\delta)$ takes $(8 + o(1))n \lg n$ operations in $\mathbf{R}$. Computing $\delta g_0 \bmod x^{2n} - 1$, hence δg_0, hence $f \bmod x^{2n}$, takes $(8 + o(1))n \lg n$ operations in $\mathbf{R}$. Computing $F_{2n}(f \bmod x^{2n})$ takes $(8 + o(1))n \lg n$ operations in $\mathbf{R}$. Computing $(f \bmod x^{2n})g_0 \bmod x^{2n} - 1$, hence $(f \bmod x^{2n})g_0$, hence the aforementioned ϵ, takes $(8 + o(1))n \lg n$ operations in $\mathbf{R}$. Computing $F_{2n}(\epsilon)$ takes $(8 + o(1))n \lg n$ operations in $\mathbf{R}$. Computing ϵg_0, hence $g \bmod x^{2n}$, takes $(8 + o(1))n \lg n$ operations in $\mathbf{R}$.

The total effort is $(60 + o(1))n \lg n$ operations in $\mathbf{R}$ to compute $f \bmod x^{2n}$, $g \bmod x^{2n}$, and $F_{2n}(f \bmod x^{2n})$ from $f \bmod x^n$, $g \bmod x^n$, and $F_n(f \bmod x^n)$. Computing $f \bmod x^n$ and $g \bmod x^n$ from scratch recursively takes $(60 + o(1))n \lg n$ operations in $\mathbf{R}$. Computing just $f \bmod x^n$ takes $(44 + o(1))n \lg n$ operations in $\mathbf{R}$.

Exponential. Consider again the problem of computing $f = \exp h$ and $g = 1/f$, given h.

Write $f_0 = f \bmod x^n$, $g_0 = g \bmod x^n$, $h_0 = h \bmod x^n$, $\delta = f_0 g_0 - 1$, and $\epsilon = (\log f_0) - h \bmod x^{2n}$. Recall that $f \bmod x^{2n} = f_0 - \epsilon f_0 \bmod x^{2n}$ and $D(\epsilon) = D(f_0)g_0 - D(h) - \delta D(h_0) \bmod x^{2n}$.

Assume that f_0, g_0, and $F_n(f_0)$ are already known. Computing $F_n(g_0)$ takes $(4 + o(1))n \lg n$ operations in $\mathbf{R}$. Computing $f_0 g_0 \bmod x^n - 1$, hence δ, takes $(4 + o(1))n \lg n$ operations in $\mathbf{R}$. Computing $F_n(D(f_0))$ takes $(4 + o(1))n \lg n$ operations in $\mathbf{R}$. Computing $D(f_0)g_0 \bmod x^n - 1$, hence $D(f_0)g_0 - D(h)$, takes $(4 + o(1))n \lg n$ operations in $\mathbf{R}$. Computing $F_{2n}(g_0)$, using $F_n(g_0)$, takes $(4 + o(1))n \lg n$ operations in $\mathbf{R}$. Computing $F_{2n}(f_0)$, using $F_n(f_0)$, takes $(4 + o(1))n \lg n$ operations in $\mathbf{R}$. Computing $F_{2n}(\delta)$ takes $(8 + o(1))n \lg n$ operations in $\mathbf{R}$. Computing $F_{2n}(D(h_0))$ takes $(8 + o(1))n \lg n$ operations in $\mathbf{R}$. Computing $\delta D(h_0) \bmod x^{2n} - 1$, hence $\delta D(h_0)$, hence $D(\epsilon)$, hence ϵ, takes $(8 + o(1))n \lg n$ operations in $\mathbf{R}$. Computing $F_{2n}(\epsilon)$ takes $(8 + o(1))n \lg n$ operations in $\mathbf{R}$. Computing $\epsilon f_0 \bmod x^{2n} - 1$, hence ϵf_0, hence $f \bmod x^{2n}$, takes $(8 + o(1))n \lg n$ operations in $\mathbf{R}$. Computing $F_{2n}(f \bmod x^{2n})$ and $g \bmod x^{2n}$ takes $(32 + o(1))n \lg n$ operations in $\mathbf{R}$.

The total effort is $(96 + o(1))n \lg n$ operations in $\mathbf{R}$ to compute $f \bmod x^{2n}$, $g \bmod x^{2n}$, and $F_{2n}(f \bmod x^{2n})$ from $f \bmod x^n$, $g \bmod x^n$, and $F_n(f \bmod x^n)$. Computing $f \bmod x^n$ and $g \bmod x^n$ from scratch recursively takes $(96+o(1))n \lg n$ operations in $\mathbf{R}$. Computing just $f \bmod x^n$ takes $(80+o(1))n \lg n$ operations in $\mathbf{R}$.

5. Cross-iteration redundancy

This section explains how to squeeze a small amount of extra performance out of some of the algorithms of Section 4, by considering several Newton steps simultaneously.

I'm not happy with the algorithms in this section. They have huge $o(1)$'s; they are slower than the algorithms of Section 4 for any reasonable value of n. They're messy; I don't have a simple explanation of how they produce better constants. They don't use the technique of Section 3, or FFT doubling from Section 4; I don't see how to achieve better constants by combining the techniques.

Reciprocal. Consider again the problem of computing $g = 1/f$ given f. Write f as $f_0 + f_1 x^n + f_2 x^{2n} + \dots$, where each f_i has degree below n. Similarly write g as $g_0 + g_1 x^n + g_2 x^{2n} + \dots$.

Let d be a positive integer. Assume that

$$\begin{aligned} &g_0, \dots, g_{d-1},\\ &F_{2n}(g_0), \dots, F_{2n}(g_{d-1}),\\ &F_{2n}(f_0), \dots, F_{2n}(f_{d-1}) \end{aligned}$$

are already known.

Compute $F_{2n}(f_d), \dots, F_{2n}(f_{2d-1})$. This uses $d(8 + o(1))n \lg n$ operations in $\mathbf{R}$.

Next compute $f_{d-1}g_0+\dots+f_0 g_{d-1}$, $f_d g_0+\dots+f_1 g_{d-1}$, $\dots$, $f_{2d-1}g_0+\dots+f_d g_{d-1}$. This computation uses $(d + 1)(8 + o(1))n \lg n$ operations in $\mathbf{R}$ for the inversions of F_{2n}. It also uses $O(d^2 n)$ operations in $\mathbf{R}$ for $O(d^2)$ multiplications and additions in $\mathbf{C}^{2n}$, but this is negligible if $d \in o(\lg n)$.

Next compute $\epsilon = (f(g \bmod x^{dn}) - 1) \bmod x^{2dn}$ by extracting the top dn coefficients of $(f_{d-1}g_0 + \dots + f_0 g_{d-1})x^{(d-1)n} + (f_d g_0 + \dots + f_1 g_{d-1})x^{dn} + \dots + (f_{2d-1}g_0 + \dots + f_d g_{d-1})x^{(2d-1)n} \bmod x^{2dn}$. This sum differs from ϵ by $f_0 g_0 + \dots + (f_{d-2}g_0 + \dots + f_0 g_{d-2})x^{(d-2)n}$, which is a polynomial of degree below dn.

Next write ϵ as $\epsilon_d x^{dn} + \dots + \epsilon_{2d-1}x^{(2d-1)n}$, and compute $F_{2n}(\epsilon_i)$ for each i. This uses $d(8 + o(1))n \lg n$ operations in $\mathbf{R}$.

Next compute $\epsilon_d g_0$, $\epsilon_{d+1}g_0 + \epsilon_d g_1$, $\dots$, $\epsilon_{2d-1}g_0 + \dots + \epsilon_d g_{d-1}$. This uses $d(8 + o(1))n \lg n$ operations in $\mathbf{R}$ if $d \in o(\lg n)$.

Next compute $g \bmod x^{2dn} = (g \bmod x^{dn}) - \epsilon(g \bmod x^{dn}) \bmod x^{2dn} = g_0 + g_1 x^n + \dots + g_{d-1}x^{(d-1)n} - (\epsilon_d g_0)x^{dn} - (\epsilon_{d+1}g_0 + \epsilon_d g_1)x^{(d+1)n} - \dots - (\epsilon_{2d-1}g_0 + \dots + \epsilon_d g_{d-1})x^{(2d-1)n} \bmod x^{2dn}$. This is fast.

Finally, compute $F_{2n}(g_d), \dots, F_{2n}(g_{2d-1})$. This uses $d(8 + o(1))n \lg n$ operations in $\mathbf{R}$.

The total effort to move from nd to $2nd$ is $(5d + 1)(8 + o(1))n \lg n$ operations in $\mathbf{R}$ if $d \in o(\lg n)$. If d is allowed to grow slowly with n then this algorithm, applied recursively, reaches n using $(40 + o(1))n \lg n$ operations in $\mathbf{R}$. Computing $g \bmod x^n$ takes only $(36 + o(1))n \lg n$ operations in $\mathbf{R}$, since $F_{2n}(g_d), \dots, F_{2n}(g_{2d-1})$ need not be computed in the last iteration.

Quotient or logarithm. As in Section 4, the first quotient strategy takes $(20+o(1))n\lg n$ operations in $\mathbf{R}$ after a reciprocal, and the second quotient strategy saves $(4+o(1))n\lg n$ operations by not computing the reciprocal.

Square root. My best algorithm along these lines uses $(44+o(1))n\lg n$ operations in $\mathbf{R}$; the much simpler algorithm of Section 4 also uses $(44+o(1))n\lg n$ operations in $\mathbf{R}$.

Exponential. Consider again the problem of computing $f=\exp h$ and $g=1/f$, given h. Write f as $f_0+f_1x^n+f_2x^{2n}+\dots$; write g as $g_0+g_1x^n+g_2x^{2n}+\dots$; write h as $h_0+h_1x^n+h_2x^{2n}+\dots$.

Let d be a positive integer. Assume that

$$\begin{aligned}
&f_0,\dots,f_{d-1},\\
&g_0,\dots,g_{d-1},\\
&F_{2n}(f_0),\dots,F_{2n}(f_{d-1}),\\
&F_{2n}(g_0),\dots,F_{2n}(g_{d-1}),\\
&F_{2n}(D(f_0)),\dots,F_{2n}\left(\frac{D(f_{d-1}x^{(d-1)n})}{x^{(d-1)n}}\right),\\
&F_{2n}(D(h_0)),\dots,F_{2n}\left(\frac{D(h_{d-1}x^{(d-1)n})}{x^{(d-1)n}}\right),
\end{aligned}$$

are already known.

Compute $f_{d-1}g_0+\dots+f_0g_{d-1}$, $f_{d-1}g_1+\dots+f_1g_{d-1}$, …, $f_{d-1}g_{d-1}$. This uses $d(8+o(1))n\lg n$ operations in $\mathbf{R}$.

Next compute $\delta=(f \bmod x^{dn})(g \bmod x^{dn})-1$ by extracting the top dn coefficients of $(f_{d-1}g_0+\dots+f_0g_{d-1})x^{(d-1)n}+(f_{d-1}g_1+\dots+f_1g_{d-1})x^{dn}+\dots+(f_{d-1}g_{d-1})x^{(2d-2)n} \bmod x^{2dn}$. This sum differs from δ by $f_0g_0+\dots+(f_{d-2}g_0+\dots+f_0g_{d-2})x^{(d-2)n}$, which is a polynomial of degree below dn.

Next write δ as $\delta_dx^{dn}+\dots+\delta_{2d-1}x^{(2d-1)n}$, and compute $F_{2n}(\delta_i)$ for each i. This uses $d(8+o(1))n\lg n$ operations in $\mathbf{R}$.

Next compute

$$\begin{aligned}
&g_0\frac{D(f_{d-1}x^{(d-1)n})}{x^{(d-1)n}}+\dots+g_{d-1}D(f_0)\\
&g_1\frac{D(f_{d-1}x^{(d-1)n})}{x^{(d-1)n}}+\dots+g_{d-1}\frac{D(f_1x^n)}{x^n}-\delta_dD(h_0),\\
&g_2\frac{D(f_{d-1}x^{(d-1)n})}{x^{(d-1)n}}+\dots+g_{d-1}\frac{D(f_2x^{2n})}{x^{2n}}-\delta_{d+1}D(h_0)-\delta_d\frac{D(h_1x^n)}{x^n},\\
&\dots,\\
&g_{d-1}\frac{D(f_{d-1}x^{(d-1)n})}{x^{(d-1)n}}-\delta_{2d-2}D(h_0)-\dots-\delta_d\frac{D(h_{d-2}x^{(d-2)n})}{x^{(d-2)n}},\\
&-\delta_{2d-1}D(h_0)-\dots-\delta_d\frac{D(h_{d-1}x^{(d-1)n})}{x^{(d-1)n}}.
\end{aligned}$$

This uses $(d+1)(8+o(1))n\lg n$ operations in $\mathbf{R}$.

Next multiply these results by $x^{(d-1)n},x^{dn},\dots,x^{(2d-1)n}$, add modulo x^{2dn}, and extract the top dn coefficients. The result is exactly $D(f \bmod x^{dn})(g \bmod x^{dn})-$

$D(h) - \delta D(h \bmod x^{dn}) \bmod x^{2dn}$: the difference is a polynomial of degree below dn.

Next compute $\epsilon = \log(f \bmod x^{dn}) - h \bmod x^{2dn}$ from the formula $D(\epsilon) = D(f \bmod x^{dn})(g \bmod x^{dn}) - D(h) - \delta D(h \bmod x^{dn}) \bmod x^{2dn}$.

Next write ϵ as $\epsilon_d x^{dn} + \cdots + \epsilon_{2d-1} x^{(2d-1)n}$, and compute $F_{2n}(\epsilon_i)$ for each i. This uses $d(8 + o(1))n \lg n$ operations in $\mathbf{R}$.

Next compute $f \bmod x^{2dn} = (f \bmod x^{dn}) - \epsilon(f \bmod x^{dn}) \bmod x^{2n}$. This uses $d(8 + o(1))n \lg n$ operations in $\mathbf{R}$.

Next compute $F_{2n}(f_d), \ldots, F_{2n}(f_{2d-1})$. This uses $d(8+o(1))n \lg n$ operations in $\mathbf{R}$.

Next compute $g \bmod x^{2dn}$ as in the above algorithm for reciprocals. This uses $(3d + 1)(8 + o(1))n \lg n$ operations in $\mathbf{R}$.

Finally, compute

$$\begin{aligned}
&F_{2n}(g_d), \ldots, F_{2n}(g_{2d-1}),\\
&F_{2n}\left(\frac{D(f_d x^{dn})}{x^{dn}}\right), \ldots, F_{2n}\left(\frac{D(f_{2d-1} x^{(2d-1)n})}{x^{(2d-1)n}}\right),\\
&F_{2n}\left(\frac{D(h_d x^{dn})}{x^{dn}}\right), \ldots, F_{2n}\left(\frac{D(h_{2d-1} x^{(2d-1)n})}{x^{(2d-1)n}}\right).
\end{aligned}$$

This uses $3d(8 + o(1))n \lg n$ operations in $\mathbf{R}$.

The total effort to move from nd to $2nd$ is $(12d+2)(8+o(1))n \lg n$ operations in $\mathbf{R}$. This algorithm, applied recursively, reaches n using $(96+o(1))n \lg n$ operations in $\mathbf{R}$. Computing $f \bmod x^n$ and $g \bmod x^n$ takes only $(84 + o(1))n \lg n$ operations in $\mathbf{R}$, since the final step of the last iteration can be skipped. Computing just $f \bmod x^n$ takes only $(68 + o(1))n \lg n$ operations in $\mathbf{R}$.

References

[1] David H. Bailey, *The computation of π to 29,360,000 decimal digits using Borweins' quartically convergent algorithm*, Mathematics of Computation **50** (1988), 283–296. ISSN 0025–5718. MR 88m:11114. Available from `http://cr.yp.to/bib/entries.html#1988/bailey`.

[2] Daniel J. Bernstein, *Detecting perfect powers in essentially linear time*, Mathematics of Computation **67** (1998), 1253–1283. ISSN 0025–5718. MR 98j:11121. Available from `http://cr.yp.to/papers.html`.

[3] Daniel J. Bernstein, *Fast multiplication and its applications*, to appear in Buhler-Stevenhagen *Algorithmic number theory* book. Available from `http://cr.yp.to/papers.html#multapps`.

[4] Daniel J. Bernstein, *Computing logarithm intervals with the arithmetic-geometric-mean iteration*. Available from `http://cr.yp.to/papers.html#logagm`. ID `8f92b1e3ec7918d37b28b9efcee5e97f`.

[5] Richard P. Brent, *Multiple-precision zero-finding methods and the complexity of elementary function evaluation*, in [12] (1976), 151–176. MR 54:11843. Available from `http://web.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub028.html`.

[6] Stephen A. Cook, *On the minimum computation time of functions*, Ph.D. thesis, Department of Mathematics, Harvard University, 1966. Available from `http://cr.yp.to/bib/entries.html#1966/cook`.

[7] Alan H. Karp, Peter Markstein, *High-precision division and square root*, Technical Report HPL-93-42(R.1), 1994; see also newer version [8]. Available from `http://www.hpl.hp.com/techreports/93/HPL-93-42.html`.

[8] Alan H. Karp, Peter Markstein, *High-precision division and square root*, ACM Transactions on Mathematical Software **23** (1997), 561–589; see also older version [7]. ISSN 0098–3500. MR 1 671 702. Available from `http://www.hpl.hp.com/personal/Alan_Karp/publications/publications.html`.

[9] H. T. Kung, *On computing reciprocals of power series*, Numerische Mathematik **22** (1974), 341–348. ISSN 0029–599X. MR 50:3536. Available from `http://cr.yp.to/bib/entries.html#1974/kung`.

[10] Arnold Schönhage, Andreas F. W. Grotefeld, Ekkehart Vetter, *Fast algorithms: a multitape Turing machine implementation*, Bibliographisches Institut, Mannheim, 1994. ISBN 3–411–16891–9. MR 96c:68043.

[11] Malte Sieveking, *An algorithm for division of powerseries*, Computing **10** (1972), 153–156. ISSN 0010–485X. MR 47:1257. Available from `http://cr.yp.to/bib/entries.html#1972/sieveking`.

[12] Joseph F. Traub, *Analytic computational complexity*, Academic Press, New York, 1976. MR 52:15938.

Department of Mathematics, Statistics, and Computer Science (M/C 249), The University of Illinois at Chicago, Chicago, IL 60607–7045

E-mail address: `djb@pobox.com`