A FAST ALGORITHM FOR REVERSION OF POWER SERIES

FREDRIK JOHANSSON

ABSTRACT. We give an algorithm for reversion of formal power series, based on an efficient way to implement the Lagrange inversion formula. Our algorithm requires $O(n^{1/2}(M(n) + MM(n^{1/2})))$ operations where $M(n)$ and $MM(n)$ are the costs of polynomial and matrix multiplication, respectively. This matches the asymptotic complexity of an algorithm of Brent and Kung, but we achieve a constant factor speedup whose magnitude depends on the polynomial and matrix multiplication algorithms used. Benchmarks confirm that the algorithm performs well in practice.

1. INTRODUCTION

Classical algorithms for composition and reversion of power series truncated to order n require $O(n^3)$ operations on elements in the coefficient ring [13]. This can be improved to $O(nM(n))$ where $M(n)$ is the cost of multiplying polynomials of degree less than n. In [5], Brent and Kung gave two asymptotically faster algorithms for composition, and observed that any algorithm for composition can be used for reversion (and vice versa) via Newton iteration, with at most a constant factor slowdown.

The first algorithm (BK 2.1) requires $O(n^{1/2}(M(n) + MM(n^{1/2})))$ operations where $MM(n)$ is the complexity of multiplying two $n \times n$ matrices. This reduces to $O(n^{1/2}M(n) + n^2)$ with classical matrix multiplication, and $O(n^{1/2}M(n) + n^{1.91})$ with the Strassen algorithm. The last term can be improved to $O(n^{1.688})$ with the Coppersmith-Winograd algorithm [18], or $O(n^{1.68632})$ with the recent bound for $MM(n)$ by Stothers [16] and Vassilevska Williams [19]. The best available bound for the last term is $O(n^{1.667})$, obtained by Huang and Pan [11] using improved techniques for multiplication of nonsquare matrices.

The second algorithm (BK 2.2) requires $O((n \log n)^{1/2}M(n))$ operations. This is asymptotically slower than BK 2.1 when classical $(M(n) = O(n^2))$ or Karatsuba multiplication $(M(n) = O(n^{\log_2 3}) = O(n^{1.585}))$ is used, but faster when FFT polynomial multiplication $(M(n) = O(n \log^{1+o(1)} n))$ is available [3, 18].

As noted by Brent and Kung, many special left-compositions, including the evaluation of reciprocals, square roots, and elementary transcendental functions of power series, can be done in merely $O(M(n))$ operations. Recent research has focused on speeding up such evaluations by constant factors by eliminating redundancy from Newton iteration [2, 7, 9]. Algorithms with quasi-linear complexity are also known for certain right-compositions, including right-composition by algebraic functions [17] and some special functions [4]. Improved composition algorithms over special rings have also been investigated [1, 12, 15]. However, the algorithms

of Brent and Kung have remained the best known for composition and reversion in the general case.

In this paper, we give a new algorithm for reversion analogous to BK 2.1 and likewise requiring $O(n^{1/2}(M(n) + MM(n^{1/2})))$ operations, but achieving a constant factor speedup. The speedup ratio depends on the asymptotics of $M(n)$ and $MM(n)$ and is in the range between 1.2 and 2.6 with polynomial and matrix multiplication algorithms used in practice. Our algorithm also allows incorporating the complexity refinement of Huang and Pan.

Whereas BK 2.1 can be viewed as a baby-step giant-step version of Horner's rule, our algorithm can be viewed as a baby-step giant-step version of the Lagrange inversion formula, avoiding Newton iteration entirely (apart from a single $O(M(n))$ reciprocal computation). It is somewhat surprising that such an algorithm has been overlooked until now, with all publications following Brent and Kung apparently having taken Newton iteration as the final word on the subject matter.

2. The algorithm

Our setting is the ring of truncated power series $R[[x]]/\langle x^n \rangle$ over a commutative coefficient ring R in which the integers $1, \ldots, n-1$ are cancellable (i.e., nonzero and not zero divisors). For example, we may take $R = \mathbb{Z}$ or $R = \mathbb{Z}/p\mathbb{Z}$ with prime $p \geq n$. We recall the Lagrange inversion formula ([13], p. 527). If $f(x) = x/h(x)$ where $h(0)$ is a unit in R, then the compositional inverse or reversion $f^{-1}(x)$ satisfying $f(f^{-1}(x)) = f^{-1}(f(x)) = x$ exists and its coefficients are given by

$$[x^k]f^{-1}(x) = \frac{1}{k}[x^{k-1}]h(x)^k.$$

The straightforward way to evaluate n terms of $f^{-1}(x)$ with the Lagrange inversion formula is to compute $h(x)$ (this requires $O(M(n))$ operations with Newton iteration) and then compute the powers $h^2, h^3, \ldots$ successively, for a total of $(n + O(1))M(n)$ operations.

Our observation is that it is redundant to compute all the powers of h given that we only are interested in a single coefficient from each. To do better, we choose some $1 \leq m < n$ and precompute $h, h^2, h^3, \ldots, h^m$. For $0 \leq k < n$, we can then write h^k as h^{i+j} where $0 \leq j < m$ and $i = lm$ for some $0 \leq l \leq \lceil n/m \rceil$, only requiring $h^m, h^{2m}, h^{3m}, \ldots$ to be computed subsequently. Determining a single coefficient in $h^k = h^i h^j$ can then be done in $O(n)$ operations using the definition of the Cauchy product. Picking $m \approx n^{1/2}$ minimizes the number of polynomial multiplications required.

We give a detailed account of this procedure as Algorithm 1. We note that most of the polynomial arithmetic is done to length $n - 1$ rather than length n, as the initial coefficient is always zero.

An improved version. Algorithm 1 clearly requires $O(n^{1/2}M(n) + n^2)$ operations in R, as many as BK 2.1 with classical matrix multiplication. We can improve the complexity by packing the inner loops into a single matrix product as shown in Algorithm 2. This allows us to exploit fast matrix multiplication.

In the description of Algorithm 2, the matrices are indexed from 1 and the pseudocode has been simplified by letting the exponents run out of bounds, using the convention that $[x^k]p = 0$ when $k < 0$ or $k \geq n - 1$. To see that the algorithm

---

Algorithm 1 Fast Lagrange inversion

Input: $f = a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1}$ where $n > 1$ and a_1 is a unit in R
Output: $g = b_1 x + \ldots + b_{n-1} x^{n-1}$ such that $f(g(x)) = g(f(x)) = x \bmod x^n$
  $m \leftarrow \lceil \sqrt{n-1} \rceil$
  $h \leftarrow x/f \bmod x^{n-1}$
  for $1 \leq i < m$ **do**
    $h^{i+1} \leftarrow h^i \times h \bmod x^{n-1}$
    $b_i \leftarrow \frac{1}{i}[x^{i-1}]h^i$
  end for
  $t \leftarrow h^m$
  for $i = m, 2m, 3m, \ldots, lm < n$ **do**
    $b_i \leftarrow \frac{1}{i}[x^{i-1}]t$
    for $1 \leq j < m$ while $i + j < n$ **do**
      $b_{i+j} \leftarrow \frac{1}{i+j} \sum_{k=0}^{i+j-1} ([x^k]t) \cdot ([x^{i+j-k-1}]h^j)$
    end for
    $t \leftarrow t \times h^m \bmod x^{n-1}$
  end for
  return $b_1 + b_2 x + \ldots + b_{n-1} x^{n-1}$

Algorithm 2 Fast Lagrange inversion, matrix version

Input: $f = a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1}$ where $n > 1$ and a_1 is a unit in R
Output: $g = b_1 x + \ldots + b_{n-1} x^{n-1}$ such that $f(g(x)) = g(f(x)) = x \bmod x^n$
  $m \leftarrow \lceil \sqrt{n-1} \rceil$
  $h \leftarrow x/f \bmod x^{n-1}$
  Assemble $m \times m^2$ matrices B and A from $h, h^2, \ldots, h^m$ and $h^m, h^{2m}, h^{3m}, \ldots$.
  for $1 \leq i \leq m, \ 1 \leq j \leq m^2$ **do**
    $B_{i,j} \leftarrow [x^{i+j-m-1}] \, h^i$
    $A_{i,j} \leftarrow [x^{im-j}] \, h^{(i-1)m}$
  end for
  $C \leftarrow AB^T$
  for $1 \leq i < n$ **do**
    $b_i \leftarrow C_i/i$ (C_i is the ith entry of C read rowwise)
  end for
  return $b_1 + b_2 x + \ldots + b_{n-1} x^{n-1}$

---

is correct, write $[x^{i_1+(i_2-1)m-1}]h^{i_1+(i_2-1)m}$ as

$$\sum_{j=0}^{i_1+(i_2-1)m-1} \left([x^j]\, h^{i_1}\right) \left(\left[x^{i_1+(i_2-1)m-1-j}\right] h^{(i_2-1)m}\right)$$

and shift the summation index to obtain

$$\sum_{j=m-i_1+1}^{i_2 m} \left([x^{i_1+j-m-1}]\, h^{i_1}\right) \left([x^{i_2 m-j}]\, h^{(i_2-1)m}\right)$$

which is the inner product of the nonzero part of row i_1 in B with the nonzero part of row i_2 in A.

The structure of the matrices is perhaps illustrated more clearly by an example. Taking $n = 8$ and $m = 3$, we need the coefficients of $1, x, \ldots, x^6$ in powers of h. Letting h_i^k denote $[x^i]h^k$, the matrices become

$$A = \begin{pmatrix} h_2^0 & h_1^0 & h_0^0 & 0 & 0 & 0 & 0 & 0 & 0 \\ h_5^3 & h_4^3 & h_3^3 & h_2^3 & h_1^3 & h_0^3 & 0 & 0 & 0 \\ (h_8^6) & (h_7^6) & h_6^6 & h_5^6 & h_4^6 & h_3^6 & h_2^6 & h_1^6 & h_0^6 \end{pmatrix},$$

$$B = \begin{pmatrix} 0 & 0 & h_0^1 & h_1^1 & h_2^1 & h_3^1 & h_4^1 & h_5^1 & h_6^1 \\ 0 & h_0^2 & h_1^2 & h_2^2 & h_3^2 & h_4^2 & h_5^2 & h_6^2 & (h_7^2) \\ h_0^3 & h_1^3 & h_2^3 & h_3^3 & h_4^3 & h_5^3 & h_6^3 & (h_7^3) & (h_8^3) \end{pmatrix},$$

where entries in parentheses do not contribute to the final result and may be set to zero. In this example the coefficient of x^4 in h^5 is given by the fifth entry in C, namely $C_{2,2} = h_4^3 h_0^2 + h_3^3 h_1^2 + h_2^3 h_2^2 + h_1^3 h_3^2 + h_0^3 h_4^2$.

3. COMPLEXITY ANALYSIS

We now study the complexity in some more detail. Let $m = \lceil \sqrt{n-1} \rceil$. Then Algorithm 2 involves at most:

(1) $2m + O(1)$ polynomial multiplications, each with cost $M(n)$,
(2) one $(m \times m^2)$ times $(m^2 \times m)$ matrix multiplication,
(3) $O(n)$ additional operations.

For comparison, BK 2.1 requires at most:

(1) m polynomial multiplications, each with cost $M(n)$,
(2) one $(m \times m)$ times $(m \times m^2)$ matrix multiplication,
(3) m polynomial multiplications and additions, each with cost $M(n) + n$.

Brent and Kung break the matrix multiplication into m products of $m \times m$ matrices, requiring $mMM(m)$ operations. We can do the same in Algorithm 2, writing the product as a length-m inner product of $m \times m$ matrices. The extra $O(n^{3/2})$ additions in this matrix operation do not affect the complexity, but it is interesting to note that they match the $O(n^{3/2})$ additions in the last polynomial stage of BK 2.1. To summarize, both Algorithm 2 and BK 2.1 require at most $(2n^{1/2} + O(1))M(n) + n^{1/2}MM(n^{1/2}) + O(n^{3/2})$ operations.

The primary drawback of our algorithm as well as BK 2.1 is the requirement to store $O(n^{3/2})$ temporary coefficients in memory, compared to $O(n \log n)$ for BK 2.2 and $O(n)$ for a naive implementation of Lagrange inversion.

Avoiding Newton iteration. In effect, we need the same number of operations to perform a length-n reversion with fast Lagrange inversion as to perform a length-n composition with BK 2.1. However, to perform a reversion with BK 2.1, we must employ Newton iteration. Using the update

$$g_{k+1}(x) = \frac{f(g_k(x)) - x}{f'(g_k(x))},$$

where the chain rule allows us to reuse the composition in the numerator for the denominator, this entails computing a sequence of compositions of length $l = 1, \ldots, \lceil n/4 \rceil, \lceil n/2 \rceil, n$, plus a fixed number of polynomial multiplications of the

same length at each stage. If c and r are such that a length-n composition takes $C(n) = cn^r$ operations, Newton iteration asymptotically takes

$$C(n) + C(n/2) + C(n/4) + \ldots = cn^r \left(\frac{2^r}{2^r - 1} \right)$$

operations, ignoring additional polynomial multiplications. For example, with classical polynomial multiplication as the dominant cost ($r = 5/2$), the speedup given by the expression in parentheses is $\frac{4}{31}(8 + \sqrt{2}) \approx 1.214$. With FFT polynomial multiplication, and classical matrix multiplication as the dominant cost ($r = 2$), the speedup is $4/3$. We note that a more efficient form of the Newton iteration might exist, in which case the speedup would be smaller.

Improving the matrix multiplication. If the matrix multiplication dominates, we can gain an additional speedup from the fact that the ith $m \times m$ block of the matrix A only has $m - i + 1$ nonzero rows, whereas the matrices in BK 2.1 are full. Classically, this gives a twofold speedup, reflected in the loop boundaries of Algorithm 1. We should ideally modify Algorithm 2 to include this saving.

In fact, a speedup is attainable with any square matrix multiplication algorithm $MM(m) \sim m^\omega$ where $\omega > 2$. For simplicity, assume that m is sufficiently composite. Do the first $m/2$ products as full products of size m, the next $(m/2 - m/3)$ in blocks of size $m/2$, the next $(m/3 - m/4)$ in blocks of size $m/3$, and so on. At stage k, only k^2 products of blocks of size m/k are required. The speedup achieved through this procedure is

$$m^{\omega+1} \left(\sum_{k=1}^{\infty} \left(\frac{m}{k} - \frac{m}{k+1} \right) k^2 \left(\frac{m}{k} \right)^\omega \right)^{-1} > \left(\sum_{k=0}^{\infty} \frac{2^{k-1}}{2^{k\omega}} \right)^{-1} = 2 - 2^{2-\omega} > 1,$$

where the nontrivial inequality can be obtained by considering the analogous subdivison with blocks of size $m/2^k$ only.

Alternatively, we can write $AB^T = (AP)(P^{-1}B^T)$ where P is a permutation matrix that makes each $m \times m$ block in A lower triangular, and use any algorithm that speeds up multiplication between a full and a triangular matrix. A simple recursive decomposition of size-k blocks into size-$k/2$ blocks has a proportional cost of $C(k) = 4C(k/2) + 2(k/2)^\omega + O(k^2)$, providing a speedup of $2^{\omega-1} - 2$. This is greater than 1 when $\omega > \log_2 6 \approx 2.585$, and better than the first method when $\omega > 1 + \log_2(2 + \sqrt{2}) \approx 2.771$. In particular, we recover an optimal factor-two speedup with classical multiplication, and a $3/2$ speedup with the Strassen algorithm.

Using rectangular multiplication. Let $n = m^2$. In the preceding analysis, we have multiplied $m \times n$ matrices via decomposition into square blocks. Remarkably, Huang and Pan have shown [11] that this is not asymptotically optimal with the best presently known algorithms. Letting $MM(x, y, z)$ denote the complexity of multiplying a matrix of size $x \times y$ by a matrix of size $y \times z$, Huang and Pan show that $MM(m, m, n) = O(n^{1.667})$, improving on the best known bound $mMM(m, m, m) = O(n^{1.68632})$ obtained via multiplication of square matrices.

The complexity improvement of Huang and Pan also translates to Algorithm 2. More precisely, given any algorithm for $m \times m$ by $m \times n$ matrix multiplication over a general ring, there is a transposed version for $m \times n$ by $n \times m$ matrix multiplication that uses the same number of scalar multiplications [10] and a number of extra

TABLE 1. Theoretical speedup of Algorithm 2 over BK 2.1 due
to avoiding Newton iteration and exploiting the matrix structure.
*Assuming that matrix multiplication can be ignored.

Dominant operation	Complexity	Newton	Matrix	Total
Polynomial, classical	$O(n^{5/2})$	1.214	1	1.214
Polynomial, Karatsuba	$O(n^{1/2+\log_2 3})$	1.308	1	1.308
Matrix, classical	$O(n^2)$	1.333	2.000	2.666
Matrix, classical, n-bit coeff.	$O(n^{3+o(1)})$	1.142	2.000	2.285
Matrix, Strassen	$O(n^{(1+\log_2 7)/2})$	1.364	1.500	2.047
Matrix, Cop.-Win.	$O(n^{1.688})$	1.450	1.229	1.782
Matrix, Huang-Pan	$O(n^{1.667})$	1.458	1?	1.458?
(Polynomial, FFT)*	$O(n^{3/2+o(1)})$	1.546	1	1.546
(Polynomial, FFT, n-bit coeff.)*	$O(n^{5/2+o(1)})$	1.214	1	1.214

scalar additions bounded by the number of entries [14]. We can therefore take
$MM(m, n, m) = (1 + o(1))MM(m, m, n)$.

With the Huang-Pan algorithm, we do not know whether a constant factor can be
saved by exploiting the zero entries. This problem would be interesting to explore
further. In any case, the Huang-Pan algorithm is currently only of theoretical
interest, as the advantage probably only can be realized for infeasibly large matrices.

Practical performance. Table 1 gives a summary of the theoretical speedup
gained by Algorithm 2 over BK 2.1 with various matrix and polynomial multiplica-
tion algorithms. With FFT-based polynomial multiplication, BK 2.2 is asymptot-
ically faster than BK 2.1 and hence also than Algorithm 2. In practice, however,
the overhead of quasilinear polynomial multiplication compared to matrix multi-
plication is likely to be large. Fast Lagrange inversion can therefore be expected to
be faster than not only BK 2.1 but also BK 2.2 even for quite large n.

Of course, counting ring operations may not accurately reflect actual speed since
operations in most interesting rings take a variable amount of time to execute on a
physical computer. One consequence of this fact is that Newton iteration is likely
to impose a smaller overhead than predicted, since coefficients generally are smaller
in earlier steps than in later ones. Newton iteration can also be expected to perform
better than generically when the output as a whole has small coefficients.

Over $\mathbb{Z}$, in particular, arithmetic operations with b-bit numbers cost $O(b^{1+o(1)})$,
where the complexity measure is the number of bit operations. In power series
arising in applications, we often have $b = O(n^{1\pm\varepsilon})$. Two complexity estimates
based on this assumption are included in Table 1. In practice, the speed will
be sensitive to the sizes of the coefficients appearing internally in each algorithm,
varying with the structure of $f(x)$.

We note that fast Lagrange inversion becomes faster than generically when the
coefficients of $x/f(x)$ grow slowly. This is often the case when $f(x)$ is a rational
function. The reversion of a rational function of fixed degree can be computed
faster by a dedicated algorithm (Newton iteration takes $O(M(n))$ operations, using
polynomial evaluation and series division for the composition), but it is desirable
for a general-purpose algorithm to be efficient in this common case, and Lagrange
inversion of course also works for nonrational functions having this growth property.

4. Benchmarks

We have implemented tuned versions of naive Lagrange inversion ("Lagrange"), BK 2.1 with Newton iteration, and Algorithm 1 ("Fast Lagrange") over $\mathbb{Z}/p\mathbb{Z}$, $\mathbb{Z}$ and $\mathbb{Q}$ as part of the FLINT library [8]. For each of these rings, FLINT provides fast coefficient arithmetic (using MPIR [6] for bignum arithmetic) and asymptotically fast polynomial multiplication using Kronecker substitution and the Schönhage-Strassen FFT algorithm. Matrix multiplication over $\mathbb{Z}/p\mathbb{Z}$ uses the Strassen algorithm when the smallest dimension is at least 256, which in principle helps BK 2.1 for $n > 256^2$ (the speedup is not significant for feasible n, however).

Timings over $\mathbb{Z}/p\mathbb{Z}$ obtained on an Intel Xeon E5-2650 2.0 GHz CPU with 256 GiB of RAM are given in Table 2. Algorithm 1 consistently runs about 1.6 times faster than BK 2.1, agreeing with a predicted speedup of 1.546 with quasilinear polynomial multiplication and negligible cost of matrix multiplication; we see that polynomial multiplication indeed dominates in BK 2.1 for n up to at least 10^6. We have also implemented BK 2.2 over $\mathbb{Z}/p\mathbb{Z}$, finding it to take about twice as much time as BK 2.1 in the tested range. BK 2.2 might, however, be preferable for larger n due to memory limits (with $n = 10^6$ and 64-bit coefficients, BK 2.1 uses 15 GiB of memory and fast Lagrange reversion uses 7.5 GiB of memory).

TABLE 2. Timings for reversion of a random power series over $\mathbb{Z}/p\mathbb{Z}, p = 2^{63} + 29$. The percentage of time spent on matrix multiplication in BK 2.1 is shown in parentheses.

n	Lagrange	BK 2.1	BK 2.2	Fast Lagrange
10	10 μs	10	18	6.1
10^2	2.8 ms	0.92	1.6	0.54
10^3	690 ms	66	120	45
10^4	110 s	3.3 (8%)	7.1	2.1
10^5	12100 s	144 (20%)	315	85
10^6	$1.9 \cdot 10^6$ s	8251 (28%)	15131	4832

Ring operations in $\mathbb{Z}$ and $\mathbb{Q}$ do not take constant time, as reflected in Tables 3 and 4. Timings are roughly cubic in n as expected from theory, but sensitive to the inputs. Fast Lagrange inversion is the fastest algorithm for small n in all examples, the fastest in all examples over $\mathbb{Z}$, and substantially faster for the rational functions f_3 and f_6 (in both cases $x/f(x)$ has small coefficients). For large n, BK 2.1 performs well on f_4 and f_5, presumably due to generating smaller coefficients internally.

With larger coefficients (as seen especially in the case of f_1), matrix multiplication appears to take a larger proportion of the time, suggesting that BK 2.2 becomes competitive for smaller n. We have not implemented BK 2.2 over $\mathbb{Z}$ and $\mathbb{Q}$, however, and can therefore not provide a direct comparison.

Care must be taken to handle denominators efficiently. In our implementation of BK 2.1, we found that naive matrix multiplication over $\mathbb{Q}$ took ten times as long as polynomial multiplications. Clearing denominators and multiplying matrices over $\mathbb{Z}$ resulted in a comparable time being spent on the matrix and polynomial stages. Similar concerns apply when implementing Algorithms 1 and 2. On the other hand, translating the *entire* composition or reversion to one over $\mathbb{Z}$ by rescaling the inputs typically results in too much coefficient inflation, and can even run slower than a

TABLE 3. Timings for the reversion of $f_1(x) = \sum_{k \geq 1} k! x^k$, $f_2(x) = \frac{x}{\sqrt{1-4x}}$, $f_3(x) = \frac{x+x^2}{1+x+x^2}$ over $\mathbb{Z}$.

n	Lagrange			BK 2.1			Fast Lagrange		
	f_1	f_2	f_3	f_1	f_2	f_3	f_1	f_2	f_3
10	7.0 µs	6.5	6.3	10	10	10	4.9	4.3	4.1
10^2	31 ms	7.2	3.2	7.8	2.1	2.1	6.4	0.96	0.65
10^3	106 s	10	4.5	10	1.1	0.96	7.1	0.71	0.22
				(38%)	(31%)	(11%)			
10^4	-	-	-	24356 s	1453	538	8903	426	152
				(81%)	(67%)	(10%)			

TABLE 4. Timings for the reversion of $f_4(x) = \exp(x) - 1$, $f_5(x) = x\exp(x)$, $f_6(x) = \frac{3x(1-x^2)}{2(1-x+x^2)^2}$ over $\mathbb{Q}$.

n	Lagrange			BK 2.1			Fast Lagrange		
	f_4	f_5	f_6	f_4	f_5	f_6	f_4	f_5	f_6
10	15 µs	15	13	31	28	28	11	11	9.1
10^2	40 ms	40	10	12	21	8.8	8.9	8.1	1.9
10^3	145 s	133	9.7	8.8	17	3.1	14	13	0.65
				(28%)	(24%)	(19%)			
10^4	-	-	-	13812 s	27057	1990	35633	27823	784
				(27%)	(27%)	(14%)			

classical algorithm working directly over $\mathbb{Q}$. We expect the situation to be similar when working with parametric power series having rational functions as coefficients.

An interesting alternative would be to work modulo small primes and reconstruct the output using the Chinese remainder theorem. We have not investigated this option in detail. It would provide additional memory benefits: for example, if the coefficients have $O(n)$ bits, direct application of BK 2.1 or fast Lagrange reversion uses $O(n^{5/2})$ bits of temporary space, while modular reversions each require $O(n^{3/2+o(1)})$ bits of space; less than the $O(n^2)$ bits required to store the output.

5. CONCLUSION

Fast Lagrange inversion is a practical algorithm for reversion of formal power series, having essentially no higher overhead than a naive implementation of Lagrange inversion for small n and requiring fewer coefficient operations than Newton iteration coupled with BK 2.1 for large n. Among currently available general-purpose algorithms, it is likely to be the fastest choice for typical coefficient rings, input series, and values of n, and may thus be a good choice as a default reversion algorithm in a computer algebra system. Newton iteration with BK 2.2 remains faster asymptotically when FFT polynomial multiplication is available, and uses less memory, but may require very large n to become advantageous.

An interesting question is whether a reversion analog of BK 2.2 can be constructed that avoids Newton iteration, or whether we can otherwise save constant factors in BK 2.2. We may also ask whether the close correspondence in complexity between Algorithm 2 and BK 2.1 can be explained by some underlying duality along the lines of the transposition principle. Further investigation of improvements over particular rings and of the special matrix multiplications arising in Algorithm 2 and BK 2.1 would also be warranted.

Acknowledgements

References

[1] Daniel J. Bernstein, *Composing power series over a finite ring in essentially linear time*, J. Symbolic Comput. **26** (1998), no. 3, 339–341, DOI 10.1006/jsco.1998.0216. MR1633872

[2] Daniel J. Bernstein, *Removing redundancy in high-precision Newton iteration*, `http://cr.yp.to/papers.html#fastnewton`, 2004.

[3] Daniel J. Bernstein, *Fast multiplication and its applications*, Algorithmic number theory: lattices, number fields, curves and cryptography, Math. Sci. Res. Inst. Publ., vol. 44, Cambridge Univ. Press, Cambridge, 2008, pp. 325–384. MR2467550 (2010a:68186)

[4] Alin Bostan, Bruno Salvy, and Éric Schost, *Power series composition and change of basis*, ISSAC 2008, ACM, New York, 2008, pp. 269–276, DOI 10.1145/1390768.1390806. MR2513515 (2010h:68221)

[5] R. P. Brent and J. T. Kung, *Fast algorithms for manipulating formal power series*, Journal of the ACM, **25** (1978), no. 4, 581–595.

[6] The MPIR development team, *MPIR: Multiple Precision Integers and Rationals*, `http://www.mpir.org`.

[7] G. Hanrot and P. Zimmermann, *Newton iteration revisited*, `http://www.loria.fr/~zimmerma/papers/fastnewton.ps.gz`, 2004.

[8] W. B. Hart, *Fast Library for Number Theory: An Introduction*, Proceedings of the Third international congress conference on Mathematical software (Berlin, Heidelberg), ICMS'10, Springer-Verlag, 2010, `http://flintlib.org`, pp. 88–91.

[9] David Harvey, *Faster algorithms for the square root and reciprocal of power series*, Math. Comp. **80** (2011), no. 273, 387–394, DOI 10.1090/S0025-5718-2010-02392-0. MR2728985 (2011m:68299)

[10] J. Hopcroft and J. Musinski, *Duality applied to the complexity of matrix multiplication and other bilinear forms*, SIAM J. Comput. **2** (1973), 159–173. MR0471439 (57 #11172)

[11] Xiaohan Huang and Victor Y. Pan, *Fast rectangular matrix multiplication and applications*, J. Complexity **14** (1998), no. 2, 257–299, DOI 10.1006/jcom.1998.0476. MR1629113 (99i:15002)

[12] Kiran S. Kedlaya and Christopher Umans, *Fast polynomial factorization and modular composition*, SIAM J. Comput. **40** (2011), no. 6, 1767–1802, DOI 10.1137/08073408X. MR2863194

[13] Donald E. Knuth, *The art of computer programming. Vol. 2*, 2nd ed., Addison-Wesley Publishing Co., Reading, Mass., 1981. Seminumerical algorithms; Addison-Wesley Series in Computer Science and Information Processing. MR633878 (83i:68003)

[14] Robert L. Probert, *On the additive complexity of matrix multiplication*, SIAM J. Comput. **5** (1976), no. 2, 187–203. MR0408311 (53 #12076)

[15] Peter Ritzmann, *A fast numerical algorithm for the composition of power series with complex coefficients*, Theoret. Comput. Sci. **44** (1986), no. 1, 1–16, DOI 10.1016/0304-3975(86)90107-6. MR858688 (88d:68043)

[16] A. J. Stothers, *On the complexity of matrix multiplication*, Ph.D. thesis, University of Edinburgh, 2010.

[17] Joris van der Hoeven, *Relax, but don't be too lazy*, J. Symbolic Comput. **34** (2002), no. 6, 479–542, DOI 10.1006/jsco.2002.0562. MR1943041 (2003k:13026)

[18] Joachim von zur Gathen and Jürgen Gerhard, *Modern computer algebra*, 2nd ed., Cambridge University Press, Cambridge, 2003. MR2001757 (2004g:68202)

[19] V. Vassilevska Williams, *Breaking the Coppersmith-Winograd barrier*, `http://cs.berkeley.edu/~virgi/matrixmult.pdf`, 2011.

Research Institute for Symbolic Computation, Johannes Kepler University, 4040 Linz, Austria

E-mail address: `fredrik.johansson@risc.jku.at`