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 evaluate 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 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 length n require $O(n^3)$ operations on elements in the coefficient ring [9]. This can be improved to $O(nM(n))$ where $M(n)$ is the cost of multiplying two length-n polynomials. In [3], 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, $O(n^{1/2}M(n) + n^{1.91})$ with the Strassen algorithm, and $O(n^{1/2}M(n)+n^{1.688})$ with the Coppersmith-Winograd algorithm [11]. The last term has subsequently been improved to $O(n^{1.667})$ by Huang and Pan [8] 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.

As noted by Brent and Kung, many special compositions, including the evaluation of reciprocals, square roots, and elementary transcendental functions of power series, can be done in just $M(n)$ operations. Recent research has focused on speeding up such evaluations by constant factors by eliminating redundancy from Newton iteration [1, 5, 6]. Improved composition algorithms over special rings have also been investigated [2, 7]. 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 $MM(n)$

2010 *Mathematics Subject Classification.* Primary 68W30.
Supported by Austrian Science Fund (FWF) grant Y464-N18.

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, although the constant-factor improvement in this case becomes conjectural.

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. 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. 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.

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 ← ⌊√n⌋
h ← x/f mod x^(n−1)
for 1 ≤ i < m do
    h^(i+1) ← h^i × h mod x^(n−1)
    b_i ← (1/i)[x^(i−1)]h^i
end for
t ← h^m
for i = m, 2m, 3m, ..., lm < n do
    b_i ← (1/i)[x^(i−1)]t
    for 1 ≤ j < m while i + j < n do
        b_(i+j) ← (1/(i+j)) Σ_{k=0}^{i+j−1} ([x^k]t) · ([x^(i+j−k−1)]h^j)
    end for
    t ← t × h_m mod x^(n−1)
end for
return b_1 + b_2 x + ... + b_(n−1) x^(n−1)
```

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 $0 \le j < m$ and $i = lm$ for some $0 \le l \le \lceil n/m \rceil$, 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 always is 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.

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 $1 \le i \le m,\ 1 \le j \le m^2$ **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 $1 \le i < n$ **do**
    $b_i \leftarrow C_i/i$ (C_i is the ith entry of C read rowwise)
  end for
  return $b_1 + b_2 x + \ldots + b_{n-1} x^{n-1}$

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 $k \ge n-1$. To see that the algorithm is correct, write $[x^{i_1+(i_2-1)m-1}]h^{i_1+(i_2-1)m}$ as

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

and shift the summation index to obtain

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

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. In the preceding analysis, we have multiplied $m\times m^2$ matrices via decomposition into square blocks. Remarkably, Huang and Pan have shown [8] that this is not asymptotically optimal with the best presently known algorithms. Letting $MM(x, y, z)$ denote the complexity of multiplying an $x \times y$ matrix by a $y \times z$ matrix, Huang and Pan show that $MM(m, m, m^2) = O(n^{1.667})$, improving on the best known bound $mMM(m, m, m) = O(n^{1.688})$ obtained via multiplication of square matrices.

As the exponents of $MM(m, m^2, m)$ and $MM(m, m, m^2)$ are the same ([8], eq. 2.7), this improvement also applies to the matrix product in Algorithm 2. We are unfortunately unable to claim a constant-factor speedup in this situation, although it can plausibly be conjectured that any multiplication algorithm admits a dual version with $MM(m, m^2, m) = (1+o(1))MM(m, m, m^2)$. In any case, the improvement of Huang and Pan is currently only of theoretical interest, as the advantage probably only can be realized for infeasibly large matrices.

Table 1. Theoretical speedup of Algorithm 2 over BK 2.1 due to avoiding Newton iteration and exploiting the matrix structure.

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, 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)	$O(n^{3/2} \log^{1+o(1)} n)$	1.546	1	1.546

Practical performance. Table 1 gives a summary of the speedup gained by Algorithm 2 over BK 2.1 with various matrix and polynomial multiplication algorithms. The last row gives the speedup assuming that the cost of matrix multiplication can be ignored.

With FFT-based polynomial multiplication, BK 2.2 is asymptotically faster than BK 2.1 and hence also than Algorithm 2. In practice, however, the overhead of quasilinear polynomial multiplication compared to matrix multiplication 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.

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. Although specialized algorithms can revert rational functions even faster, 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 C library [4]. For each of these rings, FLINT provides fast coefficient arithmetic (using MPIR [10] for bignum arithmetic) and asymptotically fast polynomial multiplication using Kronecker segmentation. Strassen matrix multiplication is exploited in the implementation of BK 2.1 over $\mathbb{Z}/p\mathbb{Z}$, although this does not contribute appreciably for practical n.

Timings over $\mathbb{Z}/p\mathbb{Z}$ obtained on a Pentium T4400 2.2 GHz CPU with 3 GiB of RAM are given in Table 2. Algorithm 1 consistently runs about 1.6 times faster than BK 2.1, roughly agreeing with a predicted speedup of 1.546 with quasilinear polynomial multiplication and negligible cost of matrix multiplication. We find

TABLE 2. Timings for reversing a random power series over $\mathbb{Z}/p\mathbb{Z}, p = 2^{63} + 29$.

n	Lagrange	BK 2.1	Fast Lagrange
10	12 µs	21 µs	7.0 µs
100	3.8 ms	1.3 ms	0.76 ms
1000	950 ms	100 ms	62 ms
10000	150 s	5.1 s	3.3 s
100000	$\sim$ 6 h	260 s	160 s

TABLE 3. Timings for reversing $f_1(x) = \sum_{k\geq 1} k!x^k, f_2(x) = \frac{x}{\sqrt{1-4x}}, f_3(x) = \frac{x+x^2}{1+x+x^2}$ 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	10 µs	10	8.4	16	15	14	6.8	5.7	5.5
50	3.7 ms	1.0	0.46	1.2	0.45	0.40	1.1	0.28	0.14
100	65 ms	10	4.4	12	2.8	2.8	13	1.4	0.87
500	23 s	3.0	1.2	2.5	0.37	0.36	2.3	0.16	0.084
1000	280 s	30	11	22	2.8	2.4	18	2.0	0.54
5000	-	-	-	4100 s	340	230	2400	110	46

TABLE 4. Timings for reversing $f_4(x) = \exp(x) - 1, f_5(x) = x\exp(x), f_6(x) = \frac{3x(1-x^2)}{2(1-x+x^2)^2}$ 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	22 µs	20	19	42	44	44	15	14	13
50	5.6 ms	5.3	1.3	2.4	3.5	2.2	1.8	1.6	0.38
100	78 ms	73	12	16	27	14	17	14	2.1
500	30 s	27	3.1	2.1	3.8	1.3	3.3	2.6	0.18
1000	340 s	300	30	17	32	8.8	30	25	1.3

that the computation time in BK 2.1 indeed is dominated by polynomial multiplications rather than by matrix multiplication for feasible n. For example, matrix multiplication only takes up 10–20% of the time at both $n = 10^4$ and $n = 10^5$. This suggests (along with timings of a preliminary implementation of BK 2.2) that BK 2.2 would not be able to beat BK 2.1 in the tested range.

Over $\mathbb{Z}$ and particularly over $\mathbb{Q}$, ring operations do not take constant time and the actual performance becomes highly sensitive to the input. This is reflected in Tables 3 and 4. We observe that BK 2.1 is faster on f_4 (whose output coefficients are smaller than those of f_5) while Lagrange inversion handles the rational functions f_3 and f_6 substantially faster.

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

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. Possible directions for future research could be to identify improvements over special rings or look for constant-factor improvements in reversion via BK 2.2. Further study of the special matrix multiplications arising in Algorithm 2 and BK 2.1 would also be warranted.

References

1. D. J. Bernstein, "Removing redundancy in high-precision Newton iteration." Unpublished, 2004. `http://cr.yp.to/papers.html#fastnewton`
2. D. J. Bernstein, "Composing power series over a finite ring in essentially linear time." Journal of Symbolic Computation **26** (1998), 339–341.
3. R.P. Brent and H.T. Kung. "Fast algorithms for manipulating formal power series." Journal of the ACM **25** (1978), 581–595.
4. W. Hart, "Fast Library for Number Theory: An Introduction." Mathematical Software – ICMS 2010. *Lecture Notes in Computer Science* Vol. 6327, 88–91. `http://www.flintlib.org`
5. G. Hanrot and P. Zimmermann, "Newton iteration revisited." Unpublished, 2004. `http://www.loria.fr/~zimmerma/papers/fastnewton.ps.gz`
6. D. Harvey, "Faster algorithms for the square root and reciprocal of power series." Mathematics of Computation **80** (2011), 387–394.
7. J. van der Hoeven. "Fast composition of numeric power series." Technical Report 2008-09, Université Paris-Sud, Orsay, France, 2008. `http://www.texmacs.org/joris/fastcomp/fastcomp-abs.html`
8. X. Huang and V. Pan. "Fast rectangular matrix multiplication and applications." Journal of Complexity **14** (1998), 257–299.
9. D. Knuth. *The Art of Computer Programming, volume 2: Seminumerical algorithms*, 3rd ed., Addison-Wesley, 1998.
10. MPIR: Multiple Precision Integers and Rationals, `http://www.mpir.org`
11. J. von zur Gathen and J. Gerhard, *Modern Computer Algebra*, 2nd ed., Cambridge University Press, 2003.

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

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