Efficient Generic Quotients Using Exact Arithmetic

Stephen M. Watt
Cheriton School of Computer Science, University of Waterloo
Waterloo, Canada

ABSTRACT

The usual formulation of efficient division uses Newton iteration to compute an inverse in a related domain where multiplicative inverses exist. On one hand, Newton iteration allows quotients to be calculated using an efficient multiplication method. On the other hand, working in another domain is not always desirable and can lead to a library structure where arithmetic domains are interdependent. This paper uses the concept of a whole shifted inverse and modified Newton iteration to compute quotients efficiently without leaving the original domain. The iteration is generic to domains having a suitable shift operation, such as integers or polynomials with coefficients that do not necessarily commute.

CCS CONCEPTS

• **Theory of computation → Design and analysis of algorithms**; • **Mathematics of computing → Mathematical software**; • **Computing methodologies → Symbolic and algebraic algorithms**; **Computer algebra systems**.

KEYWORDS

quotient, remainder, integer arithmetic, polynomial arithmetic, modified Newton iteration, generic algorithms, library structure

ACM Reference Format:
Stephen M. Watt. 2023. Efficient Generic Quotients Using Exact Arithmetic. In *International Symposium on Symbolic and Algebraic Computation 2023 (ISSAC 2023).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3597066.3597076

1 INTRODUCTION

Multiple precision integer arithmetic and polynomial arithmetic lie at the heart of a number of computational fields, including computer algebra and cryptography. The most fundamental operations that cannot generally be performed in time linear in the size of the inputs are multiplication and division, *i.e.* quotient or remainder. Efficient algorithms for these operations are therefore important.

One method to perform fast integer division is to compute the inverse of the divisor to sufficient precision by Newton iteration on approximate real numbers and then obtain the quotient by multiplication. The products in the iteration step and the final one can be performed by fast multiplication to give fast division. This approach requires working in some model of the real numbers such as multiple precision floating point arithmetic, which may be undesirable. Fast computation of univariate polynomial quotients may be performed using an ideal-adic Newton iteration on reverse polynomials in $F[x]/\langle x^n \rangle$. This allows the definition of an inverse and an algebraic mechanism to drop what would be low-order terms in a direct formulation. In both the integer and polynomial cases, these methods leave the original domain. This can complicate library structure and obscure potential optimizations.

Consider the consequences of using an approximation to the divisor inverse in computing integer quotients. Basic integer operations now require a representation for approximate real numbers, either as multiple precision floating point or by some implicit mechanism. This encourages a structure where approximate and exact arithmetic are mutually dependent. Of course, extended precision floating point arithmetic libraries ultimately use integer operations, so it could be argued that they are integer computations, but the approach is significantly different. Algorithms in models of real arithmetic typically rely on values being smaller than small relative error bounds. In floating point arithmetic this is often phrased in terms of number of units in the last place, or "ulps". This is quite different than the exactness required for integer arithmetic, and totally ignores the arithmetic dynamics questions of integer iterations.

This paper presents an alternative direct iteration that can be formulated generically on rings with an efficient shift operation. Arithmetic is exact, remaining in the original ring and without increasing computational complexity. We show how an iterative method may be used to compute a "whole shifted inverse". This quantity can then be used to compute the quotient and remainder. The algorithm relies on multiplication, the method for which can be given as a parameter. Thus, even when fast multiplication relies on other abstractions, the core arithmetic library will not have a dependency.

The contributions of this paper are:

- generic whole shift and shifted inverse as basic operations,
- an in-domain iterative method for the whole shifted inverse,
- an analysis of the integer iteration properties, proofs of convergence, useful bounds and quantitative results,
- an efficient generic algorithm for quotients.

The remainder of this paper is organized as follows: We begin in Section 2 with some notational conventions, basic facts and relevant complexity results. Section 3 presents the concept of the whole shifted inverse for both integers and polynomials and introduces iterative methods based on it. Section 4 analyzes the behaviour of the integer iteration, and in particular shows where it has fixed points and the number of steps to arrive at one. Section 5 shows how these results can be applied to selecting the starting point and limit the size of intermediate computations. Section 6 combines the results of the analysis to give a complete integer algorithm. Section 7 presents the algorithm in a generic form and applied to polynomials. Finally, Section 8 provides some concluding remarks.

ISSAC 2023, July 24–27, 2023, Tromsø, Norway

ACM ISBN 979-8-4007-0039-2/23/07…$15.00
https://doi.org/10.1145/3597066.3597076

2 BACKGROUND

Notation. In addition to conventional notation, we adopt the following:

$(a\,..\,b)$, $[a\,..\,b)$, *etc*	real intervals intersected with $\mathbb{Z}$
u quo v, u rem v	quotient and remainder (see below)
$\text{prec}_B\, u$	number of base-B digits, $\lfloor \log_B \lvert u\rvert \rfloor + 1$
$\text{prec}_x\, p$	number of coefficients, $\text{degree}_x\, p + 1$
$\text{frac}\, x$	fractional part, $x - \lfloor x \rfloor$
$\text{shift}_{n,X}\, v$	whole shift (see Section 3)
$\text{shinv}_{n,X}\, v$	whole shifted inverse (see Section 3)
$X_{(i)}$	value of X at i^{th} iteration

The integer interval notation, "[a..b]" *etc*, is used by Knuth, *e.g.* [15]. The "prec" notation, abbreviating "precision", is similar to that of [16] where it is used to present certain algorithms generically for integers and polynomials. We take integers to be represented in base-B. That is, for any integer $u \neq 0$ there is $h = \text{prec}_B(u) - 1$, such that $u = \sum_{i=0}^{h} u_i B^i, \quad u_i \in \mathbb{Z},\ 0 \leq u_i < B,\ u_h \neq 0$.

Division. Given $u, v \in D$ for D an integral domain with Euclidean norm N: $D \to \mathbb{N}$, there exist quotient q and remainder r in D such that $u = q\,v + r$, $r = 0$ or $N(r) < N(v)$. For D being $\mathbb{Z}$ with N = abs or $F[x]$, F a field, with N = degree, the quotient and remainder are unique and we write $q = u$ quo v and $r = u$ rem v.

Given $u, v \in \mathbb{R}$, $v > 0$, we often use the whole quotient q and fractional remainder r, these being $q = \lfloor u/v \rfloor$ and $r = u - q\,v$. For $u, v \in \mathbb{R}$, $x_{(0)} \in (0, 2u/v)$, the value u/v for easily invertible u is the solution to $f(x) = u/x - v = 0$, computed by Newton iteration

$$x_{(i+1)} = x_{(i)} - \frac{f(x_{(i)})}{f'(x_{(i)})} = x_{(i)} + x_{(i)} \left(1 - \frac{v}{u} x_{(i)}\right), \quad x_{(i)} \in \mathbb{R}. \quad (1)$$

Algorithms. When efficient division uses multiplication the computational complexities of the two operations are intimately related. The classical algorithms for multiplication and division of N-bit integers require time $O(N^2)$ [14]. The best known upper bound for multiplication complexity is $O(N \log N)$ [9] and this is believed to be tight [1]. While this gives the best asymptotic behaviour, it is not not suitable for practical use. In practice, software libraries such as GMP [8] use different methods for different size inputs. For multiplication, typically the $O(N^2)$ classical method is used for smallest values, the Karatsuba $O(N^{\log_2 3})$ method [13] or a Toom-Cook $O(N^{\log_3 5})$ method [6] for intermediate sized values, and another method, such as the Schönhage-Strassen $O(N \log N \log\log N)$ FFT method [17], for the largest values.

Using a multiplication with complexity $O(M(N))$, polynomial division may be computed by Newton iteration in complexity at most $O(\log N\, M(N))$ or complexity $O(M(N))$ if fast multiplication is used. [4]. For integer division, working with extended precision floating point, matters are slightly more complicated due to carries and the region of convergence. Aho, Hopcroft and Ullman [2] show how an approach similar to ours may be used for integers base 2. Hitz and Kaltofen [11] show how Newton iteration may be used to compute reciprocals in residue number systems.

It is often preferable to use classical $O(N^2)$ division methods for small values and direct methods with Karatsuba complexity, such as that of Jebelean [12] or Burnikel and Ziegler [5], for integers of intermediate size. Useful overviews are given in [4, 7].

3 THE WHOLE SHIFTED INVERSE

3.1 Integer Definitions and Facts

We are interested in computing quotients and remainders of integers using ring operations, rather than in a model of the reals. We make use of an operation that is either a multiplication or a special weak quotient, that is:

Definition 1 (Whole shift in $\mathbb{Z}$). *Given integers $B > 1$, n and u, the* base-B whole n-shift of u *is*

$$\text{shift}_{n,B}(u) = \lfloor u B^n \rfloor.$$

When B is clear by context, we write $\text{shift}_n\, u$.

When $n \geq 0$, this the integer multiplication $u \times B^n$. When $n < 0$, this is a specialized quotient. If integers are given in base-B, this operation can be highly efficient, taking time $O(1)$ or $O(\log u + n)$, depending on the details of the representation.

We now define another specialized quotient, the computation of which is the main object of this article.

Definition 2 (Whole shifted inverse in $\mathbb{Z}$). *Given integers $B > 1$, $n \geq 0$ and $v \neq 0$, the* whole base-B n-shifted inverse of v with respect to B *is*

$$\text{shinv}_{n,B}(v) = \lfloor B^n / v \rfloor.$$

When B is clear by context, we write $\text{shinv}_n\, v$.

This operation generalizes the reciprocal operation of [2] to arbitrary bases, $\text{shinv}_{2\log_2 v - 1,\,2}(v) = \text{reciprocal}(v)$, and it can be used to compute general quotients in what can be viewed as a case of Barrett reduction [3, 10].

Theorem 1 (Quotient by whole shifted inverse in $\mathbb{Z}$). *Given two positive integers u and v, with $u \leq B^h$,*

$$u \text{ quo } v = \text{shift}_{-h}(u \cdot \text{shinv}_h\, v) + \delta, \quad \delta \in \{0, 1\}.$$

Proof. From the definitions and the fact that $u \leq B^h$, we have

$$\text{shift}_{-h}(u \cdot \text{shinv}_h\, v) = u/B^h(B^h/v - \epsilon_1) - \epsilon_2, \quad 0 \leq \epsilon_i < 1$$

$$\Leftrightarrow u \text{ quo } v - 2 < \text{shift}_{-h}(u \cdot \text{shinv}_h\, v) \leq u \text{ quo } v + (u \text{ rem } v)/v.$$

Since $(u \text{ rem } v)/v < 1$ and shift maps to $\mathbb{Z}$, the result follows. □

Checking all $2 \leq u \leq 10^6$, $2 \leq v \leq u$, we find that $\delta = 0$ and $\delta = 1$ occur with approximately equal frequency.

An Integer Iteration to Compute $\text{shinv}_h\, v$

We observe that if u is specialized to B^h in the Newton iteration (1), then it becomes an iteration to compute $\text{shinv}_h\, v$. This iteration requires a real division, however. This real division is close to being a shift, so we instead use the modified iteration:

$$w_{(i+1)} = w_{(i)} + \text{shift}_{-h}\left(\text{shift}_h\, w_{(i)} - v w_{(i)}^2\right), \qquad w_{(i)} \in \mathbb{Z} \quad (2)$$
$$= w_{(i)} + \left\lfloor w_{(i)} (B^h - v w_{(i)}) B^{-h} \right\rfloor$$

and show in Section 4 that the iteration gives the desired result

$$w_{(i)} \to \text{shinv}_h\, v.$$

We note that this is *not* the usual Newton iteration, as it discretized to integers. We must therefore examine its properties in order to justify our claim that it computes the whole shifted inverse.

3.2 Polynomial Definitions and Facts

We are also interested in the efficient computation of univariate polynomial quotients. A well-known method is to use Newton iteration to compute a modular inverse of a reversed polynomial. Specifically, to compute $q = u \text{ quo } v$ for $u, v \in F[x]$, let h and k be the degrees of u and v respectively, and

$$\begin{aligned} v^* &= \text{inverse of } \operatorname{rev}_k v && (\text{mod } x^{h-k+1}) \text{ by Newton iteration} \\ q^* &= \operatorname{rev}_h u \times v^* && (\text{mod } x^{h-k+1}) \\ q &= \operatorname{rev}_{h-k} q^* \end{aligned}$$

where $\operatorname{rev}_n p(x) = x^n p(1/x)$. This is detailed nicely in [18]. The use of reverse polynomials modulo x^{h-k+1} is used to drop low-order terms. This reversal trick does not work for integer quotients because carries would propagate in the wrong direction. This was the reason to formulate the integer iteration in terms of shift and shinv. These operators may be defined analogously for polynomials to give an iteration without reversals. This direct formulation has the benefit that it admits certain optimizations, as shown in Section 7.

Definition 3 (Whole shift in $R[x]$). *Given a polynomial $u = \sum_{i=0}^{h} u_i x^i \in R[x]$ and integer n, the* variable-x whole n-shift of u *is*

$$\operatorname{shift}_{n,x} u = \sum_{i+n \geq 0} u_i x^{i+n}.$$

When x is clear by context, we write $\operatorname{shift}_n u$.

Definition 4 (Whole shifted inverse in $F[x]$). *Given $n \in \mathbb{N}$ and $v \in F[x]$ with field F, the* whole n-shifted inverse of v with respect to x *is*

$$\operatorname{shinv}_{n,x} v = x^n \text{ quo } v.$$

When x is clear by context, we write $\operatorname{shinv}_n v$,

With these definitions,we have the following simple theorem.

Theorem 2 (Quotient by whole shifted inverse in $F[x]$). *Given two polynomials $u, v \in F[x]$ with field F and $0 \leq \text{degree } u \leq h$,*

$$u \text{ quo } v = \operatorname{shift}_{-h}(u \cdot \operatorname{shinv}_h v).$$

Proof. Letting $p[i]$ denote the coefficient of x^i in p, we have, for $i \in [0 \,..\, h - \text{degree } v]$,

$$\begin{aligned} (u \text{ quo } v)[i] &= \big((x^h u) \text{ quo } v\big)[i+h] = \big(x^{-h} u (x^h \text{ quo } v)\big)[i] \\ &= \big(\operatorname{shift}_{-h}(u \cdot \operatorname{shinv}_h v)\big)[i], \end{aligned}$$

giving the desired result. □

A Polynomial Iteration to Compute $\operatorname{shinv}_h v$

The polynomial iteration to compute $y = \operatorname{shinv}_h v$ corresponding to (2) takes the same form,

$$y_{(i+1)} = y_{(i)} + \operatorname{shift}_{-h}\left(\operatorname{shift}_h y_{(i)} - v {y_{(i)}}^2\right), \quad y_{(i)} \in F[x]. \tag{3}$$

This is a direct formulation of the method with reverse polynomials.

4 INTEGER ITERATION PROPERTIES

The convergence of Newton iteration on the reals is well understood. We are interested, however, in the iteration of an integer-valued function involving fractions, subtraction and rounding, so some care is required to ensure that the arithmetic dynamics do not give unexpected phenomena. We consider the two functions

$$S_{\mathbb{R}} : \mathbb{R} \to \mathbb{R} = x \mapsto x + x\left(1 - \frac{v}{u}x\right) \tag{4}$$

$$S_{\mathbb{Z}} : \mathbb{Z} \to \mathbb{Z} = w \mapsto w + \left\lfloor w\left(1 - \frac{v}{u}w\right)\right\rfloor, \quad 1 < v < u. \tag{5}$$

where it is sometimes more convenient to use these in the form

$$S_{\mathbb{R}}(x) = x\left(2 - \frac{v}{u}x\right) \tag{6}$$

$$S_{\mathbb{Z}}(w) = \lfloor S_{\mathbb{R}}(w) \rfloor. \tag{7}$$

The iteration $w_{(i+1)} = S_{\mathbb{Z}}(w_{(i)})$ gives (2) when $u = B^h$. We do not specialize u in the present analysis, however, as the properties of the integer iteration do not depend on u having any particular form.

4.1 Real Convergence

We first show the properties of $S_{\mathbb{R}}$ to provide an orientation for the study of $S_{\mathbb{Z}}$. We begin with the following simple real theorem for which the integer case is less straightforward.

Theorem 3 ($S_{\mathbb{R}}$ Fixed Points). *The function $S_{\mathbb{R}}$ has fixed points 0 and u/v and no others.*

Proof. If $S_{\mathbb{R}}(x) = x$, then

$$x + x\left(1 - \frac{v}{u}x\right) = x \Leftrightarrow x\left(1 - \frac{v}{u}x\right) = 0$$

and the result follows. □

We will make use of the following result.

Theorem 4 ($S_{\mathbb{R}}$ Iterates). *The iterates $S^i_{\mathbb{R}}$ of $S_{\mathbb{R}}$ are given by*

$$S^i_{\mathbb{R}}(x) = \frac{u}{v}\left(1 - \left(1 - \frac{v}{u}x\right)^{2^i}\right), \quad i \geq 0. \tag{8}$$

Proof. We use induction on i. When $i = 0$, equation (8) is satisfied:

$$S^0_{\mathbb{R}}(x) = x = \frac{u}{v}\left(1 - \left(1 - \frac{v}{u}x\right)^{2^0}\right).$$

If for some $i = n \geq 0$ equation (8) holds, then

$$\begin{aligned} S^{n+1}_{\mathbb{R}}(x) &= S^n_{\mathbb{R}}(x)\left(2 - \frac{v}{u}S^n_{\mathbb{R}}(x)\right) \\ &= \frac{u}{v}\left(1 - \left(1 - \frac{v}{u}x\right)^{2^n}\right) \cdot \left(1 + \left(1 + \frac{v}{u}x\right)^{2^n}\right) \\ &= \frac{u}{v}\left(1 - \left(1 - \frac{v}{u}x\right)^{2^{n+1}}\right) \end{aligned}$$

and equation (8) also holds for $i = n + 1$. □

The following theorem describes how iterates behave at all points on the real line.

Theorem 5 ($S^i_{\mathbb{R}}$ Convergence). *The sequence of iterates $S^i_{\mathbb{R}}(x), i \geq 1$ converges if and only if $x \in [0, 2u/v]$. If $x = 0$ or $x = 2u/v$, then $S^i_{\mathbb{R}}(x) = 0, i \geq 1$. If $x \in (0, 2u/v)$, the sequence converges quadratically to u/v.*

Proof. The proof is split into disjoint and exhaustive cases:

Case 1, $x = 0$ or $x = 2u/v$:
We have $S_{\mathbb{R}}(x) = 0$ so $S^i_{\mathbb{R}}(x) = 0, i \geq 1$.

Case 2, $x < 0$:
When $uv \geq 0$, we have $-x^2v/u \leq 0$ so $S_{\mathbb{R}}(x) = 2x - x^2u/v \leq 2x < 0$.
When $uv < 0$, we have $2 - xv/u > 2$ so $S_{\mathbb{R}}(x) = x(2 - xv/u) < 2x$.
In either case, $S^i_{\mathbb{R}}(x) \leq 2^i x$ and grows negatively without bound.

Case 3, $x > 0, uv \leq 0$:
Since $2x > 0$ and $-x^2v/u \geq 0$, we have $S_{\mathbb{R}}(x) = 2x - x^2v/u \geq 2x$ and $S^i_{\mathbb{R}}(x) \geq 2^i x$ and grows without bound.

Case 4, $0 < x < 2u/v, uv > 0$:
We observe that $0 < S_{\mathbb{R}}(x) \leq u/v$ in this region. To see this, note $S_{\mathbb{R}}(x)$ is a parabola with maximum $S_{\mathbb{R}}(x) = u/v$ at the vertex $x = u/v$, and value 0 at the excluded region endpoints. So $S_{\mathbb{R}}$ maps the entire region into $(0, u/v]$ and we need only consider $S^i_{\mathbb{R}}\big((1-\epsilon)u/v\big)$ with $0 \leq \epsilon < 1$. By Theorem 4,

$$S^i_{\mathbb{R}}\left((1-\epsilon)\frac{u}{v}\right) = (1-\epsilon)^{2^i}\frac{u}{v}, \quad i \geq 0$$

so $S^i_{\mathbb{R}}$ converges in this region and, moreover,

$$\frac{\left|S^{i+1}_{\mathbb{R}}(x) - u/v\right|}{\left|S^i_{\mathbb{R}}(x) - u/v\right|^\lambda} = v/u \quad \text{for } \lambda = 2.$$

Therefore $S^i_{\mathbb{R}}(x)$ converges quadratically to u/v for $0 < x < 2u/v$.

Case 5, $x > 2u/v, uv > 0$:
We have $2 - xv/u < 0$ so $S_{\mathbb{R}}(x) = x\left(2 - \frac{v}{u}x\right) < 0$ and $S^i_{\mathbb{R}}(S_{\mathbb{R}}(x))$ grows negatively without bound by case 2.

Summary:
Cases 1 and 4 show that the sequence converges to the claimed values when $x \in [0, 2u/v]$. Cases 2, 3 and 5 together show that the sequence does not converge when $x \notin [0, 2u/v]$. Note $[0, 2u/v]$ is empty when $uv < 0$. □

4.2 Integer Fixed Points

In order to study convergence of the sequence of $S_{\mathbb{Z}}$ iterates, we first show where $S_{\mathbb{Z}}$ has fixed points.

Theorem 6 ($S_{\mathbb{Z}}$ Fixed Points). *Given $1 < v < u \in \mathbb{Z}$, the fixed points of $S_{\mathbb{Z}}$ on $[0\,..\,u/v]$ are 0, 1, $\lfloor u/v \rfloor$ and, when*

$$\frac{u}{v} \in (1,4) \cup \bigcup_{j=4}^{\lfloor u/2 \rfloor} \left[j,\ j + \frac{1}{j-2} \right), \tag{9}$$

$\lfloor u/v \rfloor - 1$. These are 2, 3, or 4 distinct points, depending on the value of u/v.

Proof. The values 0, 1 and $\lfloor u/v \rfloor$ are easily seen to be fixed points:

$$\begin{aligned} S_{\mathbb{Z}}(0) &= 0 + \lfloor 0 \rfloor = 0 \\ S_{\mathbb{Z}}(1) &= 1 + \lfloor 1 - v/u \rfloor = 1 \\ S_{\mathbb{Z}}(\lfloor u/v \rfloor) &= \lfloor u/v \rfloor + \lfloor \mathrm{frac}(u/v)\big(1 - \mathrm{frac}(u/v)\, v/u\big) \rfloor = \lfloor u/v \rfloor \\ &\qquad \text{since } u > v. \end{aligned}$$

We find $S_{\mathbb{Z}}(\lfloor u/v \rfloor - 1) = \lfloor u/v \rfloor - 1$ is equivalent to

$$0 \leq B(u/v) < 1, \text{ where } B(x) = (\lfloor x \rfloor - 1)(x - \lfloor x \rfloor + 1)/x. \tag{10}$$

This is satisfied for $1 < u/v < 4$ because $B(x) \geq 0$ and $B(u/v) < 1$ is equivalent to $v/u = \mathrm{frac}(u/v)/\big(1 + \mathrm{frac}(u/v)\big)^2$. For $u/v \geq 4$, we have $B(u/v) - 1 = j - 2 - (j-1)^2 v/u$ on $[j, j+1)$, $j \in \mathbb{Z}$. Therefore $B(u/v) < 1$ on $[j, j+1)$ when $u/v \in \big[j, j + 1/(j-2)\big)$. Since $v \geq 2$, condition (10) holds exactly when (9) is satisfied, so $\lfloor u/v \rfloor - 1$ is a fixed point of $S_{\mathbb{Z}}$ if and only if (9) is satisfied.

We now show there are no other integer fixed points. Fixed points must satisfy

$$S_{\mathbb{Z}}(w) - w = 0 \Leftrightarrow \left\lfloor w\left(1 - \frac{v}{u}w\right) \right\rfloor = 0 \Leftrightarrow 0 \leq w - \frac{v}{u}w^2 < 1. \tag{11}$$

This locus lies below the parabola $E(x) = x - (v/u)\,x^2$ which is symmetric about $x = u/(2v)$, *i.e.* $E(u/(2v) - a) = E(u/(2v) + a)$, with E increasing for $x < u/(2v)$ and decreasing for $x > u/(2v)$.

We separately consider the points to the left and right of the line of symmetry. Only when $u \geq 4v$ is $[0\,..\,u/v] - \{0, 1, \lfloor u/v \rfloor - 1, \lfloor u/v \rfloor\}$ non-empty. Consider the elements $2 \leq n \leq \lfloor u/(2v) \rfloor$. Since E is increasing on this region and $u \geq 4v$,

$$E(w) \geq E(2) = 2 - \frac{v}{u}4 \geq 1,$$

showing these values do not satisfy (11) so are not fixed points. Next consider the elements $w \in D$ with $\lfloor u/(2v) \rfloor \leq w \leq \lfloor u/v \rfloor - 2$. Since E is decreasing on this region, the smallest value of $E(w)$ will be $E(\lfloor u/v \rfloor - 2)$. If $\lfloor u/v \rfloor - 2 \neq 2$, symmetry about $x = u/(2v)$ gives

$$E(\lfloor u/v \rfloor - 2) = E(u/v - \lfloor u/v \rfloor + 2) \geq E(2) \geq 1,$$

showing these values are also not fixed points. Combining the two halves, $S_{\mathbb{Z}}$ has no fixed points n with $2 \leq n \leq \lfloor u/v \rfloor - 2$.

The cardinality of the set $\{0, 1, \lfloor u/v \rfloor - 1, \lfloor u/v \rfloor\}$ will therefore be 2, 3 or 4, depend on the value of u/v and the condition (9). □

The region where $0 \leq B(u/v) < 1$ is shown in Figure 1.

Estimate 1. *Given an integer $u > 2$, the number of integer values v, $1 < v < u$, for which $S_{\mathbb{Z}}(\lfloor u/v \rfloor - 1) = \lfloor u/v \rfloor - 1$ is approximately*

$$\frac{\pi^2 - 5}{6}u.$$

Justification. We estimate the frequency with which $\lfloor u/v \rfloor - 1$ is a fixed point of $S_{\mathbb{Z}}$. The condition (9) is satisfied either when $u/v < 4$ or when u/v lies in an interval $\big[i, 1/(i-2)\big)$ out of a total of $u - 2$ values. We assume the values of $\mathrm{frac}(u/v)$ are uniformly distributed on $[j, j+1)$, and estimate the number of values of v that fall in an interval to be proportionate to the length of that interval times the number of u/v with $\lfloor u/v \rfloor \in [j-1, j)$. This gives the estimate

$$\begin{aligned} &\#\left\{ v \;\middle|\; S_{\mathbb{Z}}\left(\left\lfloor \frac{u}{v} \right\rfloor - 1\right) = \left\lfloor \frac{u}{v} \right\rfloor - 1,\ 1 < v < u \right\} \\ &\quad \approx u - \lfloor u/4 \rfloor - 1 + (u-2) \cdot \sum_{j=4}^{\lfloor u/2 \rfloor} \frac{1}{j-1} \cdot \left(\frac{1}{j-1} - \frac{1}{j} \right) \\ &\quad = u - \lfloor u/4 \rfloor - 1 + (u-2) \cdot \left(\left\lfloor \frac{u}{2} \right\rfloor^{-1} - \Psi\left(1, \left\lfloor \frac{u}{2} \right\rfloor\right) - \frac{19}{12} + \frac{\pi^2}{6} \right) \end{aligned}$$

and the result follows. Here, Ψ is the polygamma function, which makes a negligible contribution. □

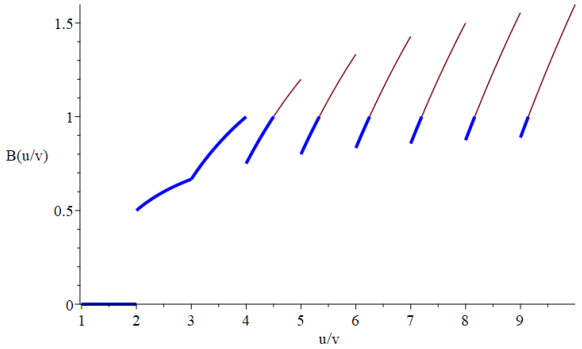

Figure 1: Region where $S_{\mathbb{Z}}(\lfloor u/v \rfloor - 1) = \lfloor u/v \rfloor - 1$ (heavy line).

u	Actual	Estimate 1	Abs Err	Rel Err
10^1	8	8	0	0
10^2	85	81	4	4.706×10^{-2}
10^3	818	811	7	8.557×10^{-3}
10^4	8,135	8,116	19	2.336×10^{-3}
10^5	81,178	81,160	18	2.217×10^{-4}
10^6	811,655	811,600	55	6.776×10^{-5}
10^7	8,116,081	8,116,007	74	9.118×10^{-6}
10^8	81,160,153	81,160,073	80	9.857×10^{-7}
10^9	811,600,878	811,600,733	145	1.787×10^{-7}
10^{10}	8,116,007,538	8,116,007,335	203	2.501×10^{-8}

Figure 2: Number of v with $S_{\mathbb{Z}}(\lfloor u/v \rfloor - 1)$ fixed point.

Figure 2 compares, for increasing u, the actual and estimated number of $1 < v < u$ making $\lfloor u/v \rfloor - 1$ a fixed point of $S_{\mathbb{Z}}$.

4.3 Integer Convergence

We now analyze the convergence of the integer iteration. On sequences of integers, convergence means arrival at a fixed point rather than entering a cycle or growing unboundedly.

Theorem 7 ($S_{\mathbb{Z}}$ Convergence). *The sequence of iterates $S^i_{\mathbb{Z}}(w), i \geq 1$ converges if and only if $w \in [0 \,..\, 2u/v]$. If the series converges, then it is to one of $\{0, 1, \lfloor u/v \rfloor - 1, \lfloor u/v \rfloor\}$. For $w \in [2 \,..\, u/v]$, $S^i_{\mathbb{Z}}(w)$ converges to $\lfloor u/v \rfloor - 1$ or $\lfloor u/v \rfloor$.*

Proof. The proof is organized in disjoint and exhaustive cases along the same lines as Theorem 5. Here, however, the cases are not independent. Their relationship is as follows: case 1b depends on 1a, 4a and 4b, case 4b depends on 4a, and the remaining cases, 1a, 2, 3 and 4a, do not depend on others.

Case 1a, $w = 0$ or 1:
If $w = 0$ or 1, we have

$$S^i_{\mathbb{Z}}(w) = w \in \{0, 1\}, i \geq 0.$$

Case 1b, $w = \lfloor 2u/v \rfloor - 1$ or $\lfloor 2u/v \rfloor$:
If $w = \lfloor 2u/v \rfloor = 2u/v - \delta_0$, then

$$S_{\mathbb{Z}}(\lfloor 2u/v \rfloor) = \left\lfloor \delta_0 \left(2 - \delta_0 \frac{v}{u}\right) \right\rfloor.$$

Since $\delta_0(2 - 2\delta_0 v/u)$ is a concave-down parabola with vertex at $\delta_0 = u/v > 1$, it is increasing for $0 \leq \delta_0 < 1$, taking values from 0 to $2 - v/u$. Since $0 < v/u < 1$, we have

$$S^i_{\mathbb{Z}}(S_{\mathbb{Z}}(\lfloor 2u/v \rfloor)) = S_{\mathbb{Z}}(\lfloor 2u/v \rfloor) \in \{0, 1\}.$$

If $w = \lfloor 2u/v \rfloor - 1 = 2u/v - \delta_1$, then

$$S_{\mathbb{Z}}(\lfloor 2u/v \rfloor - 1) = \left\lfloor 2\delta_1 + 2 - (\delta_1 + 1)^2 v/u \right\rfloor.$$

Again we have a concave-down parabola, now with vertex at $\delta_1 = u/v - 1$ which may occur inside or outside $\in [0, 1)$, depending on u/v. Considering all cases, we have

$$S_{\mathbb{Z}}(\lfloor 2u/v \rfloor - 1) \in \{0, 1, 2, 3\}.$$

The values 2 and 3 can arise only when $\lfloor 2u/v \rfloor \geq 4$. For all $\lfloor 2u/v \rfloor \geq 4$, the sequences $S^i_{\mathbb{Z}}(w)$ converge by cases 1a and 4a/b for $w \in \{0, 1, 2\}$. If $\lfloor 2u/v \rfloor > 4$, then $S^i_{\mathbb{Z}}(3)$ also converges by case 4a/b. If $\lfloor 2u/v \rfloor = 4$, then $2v \leq u < 5v/2$ so $S_{\mathbb{Z}}(3) = 1$, giving case 1a.

Case 2, $w < 0$:
We write the iteration as

$$S_{\mathbb{Z}}(w) = w + \left\lfloor w - vw^2/u \right\rfloor \leq 2w - vw^2/u < 2w$$

so $S_{\mathbb{Z}}(w) < 2^i w$ for $i \geq 0$, which grows negatively without bound.

Case 3, $w > 0, uv \leq 0$:
This does occur since $u > v > 0$.

Case 4a, $1 < w \leq \lfloor u/v \rfloor$, $uv > 0$:
On the region $(1, \lfloor u/v \rfloor - 1)$, we have $S_{\mathbb{R}}$ strictly increasing and, by Theorem 6, $S_{\mathbb{Z}}$ has no fixed points. We then have $S_{\mathbb{Z}}$ strictly increasing and

$$S_{\mathbb{Z}}(w) \geq w + 1 \quad \text{for } 1 < w < \lfloor u/v \rfloor - 1.$$

Since $S_{\mathbb{R}}(x)$ achieves its maximum value at u/v, we have $S_{\mathbb{Z}}(w) \in (1, \lfloor u/v \rfloor]$. Since the iterates increase by at least 1, there exists $i < w$ such that

$$S^i_{\mathbb{Z}}(w) \in \left(1, \lfloor u/v \rfloor\right] \setminus \left(1, \lfloor u/v \rfloor - 1\right) = \left[\lfloor u/v \rfloor - 1, \lfloor u/v \rfloor\right].$$

Therefore the sequence converges to $\lfloor u/v \rfloor - 1$ or $\lfloor u/v \rfloor$.

Case 4b, $\lfloor u/v \rfloor < w < \lfloor 2u/v \rfloor - 1$, $uv > 0$:
We assume $u/v \geq 2$, otherwise the region is empty. Since $S_{\mathbb{R}}$ is strictly decreasing on this region, $S_{\mathbb{Z}}(w)$ will be minimal at the end point $\lfloor 2u/v \rfloor - 2$. By symmetry about u/v, this is

$$\begin{aligned} S_{\mathbb{Z}}(\lfloor 2u/v \rfloor - 2) &= \lfloor S_{\mathbb{R}}(\lfloor 2u/v \rfloor - 2) \rfloor = \lfloor S_{\mathbb{R}}(2 + 2u/v - \lfloor 2u/v \rfloor) \rfloor \\ &= \lfloor (\lfloor 2u/v \rfloor - 2)(2 + 2v/u - \lfloor 2u/v \rfloor v/u) \rfloor \geq 2. \end{aligned}$$

Since $S_{\mathbb{Z}}$ is bounded above by $\lfloor u/v \rfloor$, we have $S_{\mathbb{Z}}(w) \in [2, \lfloor u/v \rfloor]$ for $\lfloor u/v \rfloor < w < \lfloor 2u/v \rfloor - 1$ and the $S^i_{\mathbb{Z}}(w)$ converges by case 4a.

Case 5, $w > \lfloor 2u/v \rfloor$, $uv > 0$:
Let $w = \lfloor 2u/v \rfloor + N = 2u/v + N - \mathrm{frac}(u/v)$ where $1 \leq N \in \mathbb{Z}$. We write the iteration as

$$S_{\mathbb{Z}}(w) = \lfloor -(N - \mathrm{frac}(u/v))\,(2 + v/u(N - \mathrm{frac}(u/v))) \rfloor < 0$$

so the sequence grows negatively without bound by Case 2. □

4.4 Initial Value and Fast Convergence

Given u and v, it is desirable to find $\lfloor u/v \rfloor$ in as few iterations as possible. To do so requires a good choice of starting value. If the starting value is too large, that is if it is larger than $\lfloor 2u/v \rfloor$, then the sequence of iterates will diverge. If it is positive, but too small, then there is another problem. Suppose that the result $\lfloor u/v \rfloor$ has b bits and the starting value has $b_0 < b$ bits. Then there will be $b - b_0$ iterations to reach an iterate with the correct length since each iteration can multiply its argument by no more than 2. That is, there will be one iteration per bit short, while we expect the usual Newton iteration to double the number of correct digits. This may be remedied with the following theorem.

Theorem 8 (Fast $S_{\mathbb{Z}}$ Convergence). *If* $w_{(0)} \in \left[(1-\frac{1}{4})\,u/v \,..\, (1+\frac{1}{4})\,u/v\right]$, $u/v \geq 2$, *then*

$$S_{\mathbb{Z}}^{\lceil \log_2 \log_2 (u/v) \rceil}(w_{(0)}) \in \left\{ \lfloor u/v \rfloor - 1, \lfloor u/v \rfloor \right\}.$$

Proof. We begin by showing the claim holds for $S_{\mathbb{R}}$. Since $u > v$ there will be at least one iteration, after which all values will be $x_{(i)} = S_{\mathbb{R}}^i(x) \leq u/v$. So we need only consider $(1-\frac{1}{4})u/v \leq x \leq u/v$. Then, for $\alpha \in \mathbb{R}$,

$$x_{(i)} = \left(1 - \frac{1}{2^{2^\alpha}}\right)\frac{u}{v} \Rightarrow x_{(i+1)} = S_{\mathbb{R}}(x_{(i)}) = \left(1 - \frac{1}{2^{2^{\alpha+1}}}\right)\frac{u}{v}.$$

When $x_{(0)}$ has the same number of bits as $\lfloor u/v \rfloor$, the number of correct leading bits of $x_{(i)}$ doubles with each iteration.

We now consider $S_{\mathbb{Z}}$. At each iteration we have

$$w_{(i+1)} = S_{\mathbb{Z}}^{i+1}(w_{(0)}) = S_{\mathbb{R}}(S_{\mathbb{Z}}^i(w_{(0)})) + \epsilon, \quad 0 \leq \epsilon < 1.$$

If $w_{(i)}$ has $\alpha_{(i)}$ correct leading bits, then $w_{(i+1)}$ will have at least $2\alpha_{(i)} - 1$ correct leading bits, so $\alpha_{(i)} = (\alpha_{(0)} - 1)2^i + 1$, with $\alpha_{(0)}$ the number of correct leading bits of $w_{(0)}$. Since $w_{(0)} \in [(1 - \frac{1}{4})u/v, (1 + \frac{1}{4}])u/v$,

$$\alpha_{(0)} \geq \log_2(4 - 1) = \log_2 3 > 2.$$

All bits will be correct if the number of iterations, i, satisfies

$$(\alpha_{(0)} - 1)2^i + 1 \geq \log_2(u/v) \tag{12}$$

so when $i \geq \lceil \log_2 \log_2(u/v) \rceil$, we have $2^i + 1 > \log_2(u/v)$ and condition (12) is satisfied. This gives the desired result. □

The condition $u/v \geq 2$ is given to avoid degenerate cases. Finding $\lfloor u/v \rfloor = 1$ is easily achieved by testing $u < v + v$.

5 INTEGER BASE AND PRECISION MATTERS

We now apply the previous results to computing $\text{shinv}_h\, v$ in base-B. There are three questions to settle. The first is how to obtain an iteration starting point efficiently that satisfies the conditions of Theorem 8. The second is how to exploit iteration accuracy to perform intermediate computations on smaller quantities. The third is to understand when a prefix of v is sufficient to compute an iterate. This section answers these questions.

5.1 Initial Value

We show a choice for the initial value of the iteration sequence that satisfy the conditions of Theorem 8. This involves inverting a short prefix of v. As the necessary prefix size is bounded, this short inversion is a constant time operation.

Theorem 9 (Initial Value Choice). *Let* $B \leq v < B^{k+1}$ *and* $2v \leq u = B^h$ *for* $B \geq 16$ *and* $v = VB^{k-f} + R$ *with* $f, V, R \in \mathbb{Z}$, $B^f \leq V < B^{f+1}$, $0 \leq R < B^{k-f}$ *and* $f \geq \min(k, 2)$. *Then the choice* $w_{(0)} = \lfloor B^{f+2}/V \rfloor B^{h-k-2}$ *gives*

$$S_{\mathbb{Z}}^{\lceil \log_2 \log_2 (u/v) \rceil}(w_{(0)}) \in \{\lfloor u/v \rfloor, \lfloor u/v \rfloor - 1\}.$$

Proof. Since $V \geq 4$ and $R < B^{k-f}$, we have $\frac{1}{4} > R/(VB^{k-f})$ so

$$\left(1 + \frac{1}{4}\right)\frac{u}{v} > \left(1 + \frac{R}{VB^{k-f}}\right)\frac{u}{v} = \frac{B^h}{VB^{k-f}} \geq \left\lfloor \frac{B^{f+2}}{V} \right\rfloor B^{h-k-2}.$$

On the other hand, since $V < B^{f+1}$ we have $B^{f+2}/(4V) > 1$ and

$$\begin{aligned}\left(1 - \frac{1}{4}\right)\frac{u}{v} &= \frac{3}{4}\frac{u}{VB^{k-f} + R} \leq \frac{3}{4}\frac{B^{h-k+f}}{V} \\ &< \left(\left\lfloor \frac{B^{f+2}}{V} \right\rfloor + 1 - \frac{B^{f+2}}{4V}\right) B^{h-k-2} < \left\lfloor \frac{B^{f+2}}{V} \right\rfloor B^{h-k-2}.\end{aligned}$$

The conditions of Theorem 8 are satisfied so we have our result. □

If $B < 16$, one may interpret the value v as base-B^p, which need not involve copying or modifying any data.

5.2 Shorter Iterates

Iterative methods to compute multiple precision values normally start with low precision then increase precision with each iteration. For variable length values, this can reduce the cost substantially. We examine how to do this when computing $\text{shinv}_h\, v$.

When computing $\text{shinv}_h(v)$ from the sequence $S_{\mathbb{Z}}^i(w_{(0)})$, only the leading digits of the intermediate iterates matter. Rather than compute a series of iterates all of full length, it is possible to compute a sequence of whole inverses, almost doubling their length at each step. Theorem 10 states this more precisely using $S_{\mathbb{Z}}$ explicitly parameterized by h and v,

$$S_{\mathbb{Z}}(h, v, w) = w + \left\lfloor w(B^h - vw)B^{-h} \right\rfloor. \tag{13}$$

Note that equation (13) is equivalent to (5) with $u = B^h$.

Theorem 10 (Shift Extension). *Let* $w = \text{shinv}_h\, v$, $B^k \leq v < B^{k+1} \leq B^h$ *and let* $w_{[n]} = \text{shift}_{n\ell-h+k}(w)$ *be the leading* $n\ell$ *digits of* w, *with* $n\ell \leq h - k$. *Then*

$$0 \leq w_{[2]} - S_{\mathbb{Z}}(k + 2\ell,\, v,\, \text{shift}_\ell\, w_{[1]}) \leq B.$$

Proof. Let $\Upsilon = S_{\mathbb{Z}}(k+2\ell,\, v,\, \text{shift}_\ell\, w_{[1]})$. Then, from the definitions and some algebra,

$$w_{[2]} - \Upsilon = \epsilon_1^2 v B^{-k} - \epsilon_2 + \epsilon_3,$$

where

$$\epsilon_1 = \text{frac}\left(B^{\ell+k}/v\right),\ \epsilon_2 = \text{frac}\left(B^{2\ell+k}/v\right),\ \epsilon_3 = \text{frac}\left(B^\ell \epsilon_1 - vB^{-k}\epsilon_1^2\right).$$

The difference $w_{[2]} - \Upsilon$ takes its largest value when v is largest, *i.e.* when $v = B^{k+1} - 1$ and

$$w_{[2]} - \Upsilon \le \epsilon_1^2 B - \epsilon_1^2 B^{-k} - \epsilon_2 + \epsilon_3 < B + 1,$$

so $w_{[2]} - \Upsilon \le B$. The difference takes its smallest value when v is smallest, *i.e.* when $v = B^k$ and

$$w_{[2]} - \Upsilon \ge \epsilon_1^2 - \epsilon_2 + \epsilon_3 \ge -\epsilon_2 > -1,$$

so $w_{[2]} - \Upsilon \ge 0$. □

Compared to the iteration of Theorem 8, this iteration may be one base-B digit short of doubling the precision at each step, rather than being one bit short. This is in trade-off against the savings from working with much smaller values. In any case, iterating (13), we have $(\ell-1)2^n+1$ correct base-B digits after n steps. It is therefore required to have $\ell \ge 2$ before starting the iteration.

It should be noted that the intermediate expression $B^h - vw$ in (13) will have about half its digits predictable in advance, since w will be a shifted inverse of w. Therefore, in principle, only about half of the digits of the product need be calculated.

5.3 Divisor Prefixes

When the divisor v is large relative to $\mathrm{shinv}_h\, v$, its lower order digits will not contribute to the shifted inverse. Even if v is not large relative to the final result, it can be large relative to the early short iterates described in Section 5.2. It is therefore interesting to see how much a divisor may be perturbed without changing the value of an iterate too much. Since Theorem 10 shows that short iterates may be one digit short of doubling the precision, we take that as the tolerance here as well. Using only a prefix of v means dropping some lower order digits, so we are interested in a negative perturbation. This is captured by the following theorem.

Theorem 11 (Divisor Sensitivity). *Let $w_{[n]}$ be as in Theorem 10 and let Δ be the decrease obtained by perturbing the divisor v by $-\delta$ in $S_{\mathbb{Z}}(k+2\ell,\, v,\, \mathrm{shift}_\ell\, w_{[1]})$, i.e.*

$$\Delta = S_{\mathbb{Z}}(k+2\ell,\, v-\delta,\, \mathrm{shift}_\ell\, w_{[1]}) - S_{\mathbb{Z}}(k+2\ell,\, v,\, \mathrm{shift}_\ell\, w_{[1]}).$$

Then

$$B^{2\ell-k-2}\delta - 1 < \Delta < B^{2\ell-k}\delta + 1. \tag{14}$$

In particular, if $\delta \le B^{k-2\ell+1}$, then $0 \le \Delta \le B$.

Proof. From the definition of $S_{\mathbb{Z}}$ and some simplification, we find

$$\Delta = \delta {w_{[1]}}^2 B^{-k} + \epsilon_1 - \epsilon_2, \tag{15}$$

where

$$\epsilon_1 = \mathrm{frac}\big(w_{[1]}\big(B^\ell - v w_{[1]} B^{-k}\big)\big),$$
$$\epsilon_2 = \mathrm{frac}\big(w_{[1]}\big(B^\ell - (v-\delta) w_{[1]} B^{-k}\big)\big).$$

Using $B^k \le v < B^{k+1}$ and $w_{[1]} = \left\lfloor \lfloor B^h/v \rfloor B^{\ell-(h-k)} \right\rfloor$, equation (15) gives (14) and if $\delta \le B^{k-2\ell+1}$, we have $\Delta < B+1$ so $0 \le \Delta \le B$. □

Theorem 11 shows that the last $k-2\ell+1$ digits of v are not required to obtain an iterate with the same order of accuracy as given by a short iterate. We may therefore adapt the iteration scheme of Theorem 10 to be

$$\begin{aligned} w_{(i+1)} &= S_{\mathbb{Z}}(k+2\ell_{(i)} - s_{(i)},\ \mathrm{shift}_{-s_{(i)}}\, v,\ \mathrm{shift}_{\ell_{(i)}}\, w_{(i)}) \\ &= w_{(i)} B^\ell + \left\lfloor w_{(i)} \left(B^\ell - B^{-k+s_{(i)}} \lfloor v B^{-s_{(i)}} \rfloor w_{(i)} \right) \right\rfloor \\ \ell_{(i+1)} &= 2\ell_{(i)} - 1 \end{aligned} \tag{16}$$

where

$$s_{(i)} = \max(0, k - 2\ell_{(i)} + 1). \tag{17}$$

5.4 Close Products

When vw is close to B^h the difference $B^h - vw$ will have many fewer than h base-B digits. When

$$\left| B^h - vw \right| \le B^e, \quad e < h, \tag{18}$$

only the lower e digits of the product vw need be computed since the upper $h-e$ digits will be determined. When $B^h > vw$, the difference will be positive and the upper digits will all be $B-1$. When $B^h < vw$, the difference will be negative and the upper digits will all be 0. When the sign of the difference is not known in advance, one may compute the lower $e+1$ digits of the product vw and the sign of the result will be given by whether the coefficient of B^e is 0 or $B-1$.

To compute the lowest e digits of vw, one need only compute the lowest e digits of $(v \text{ rem } B^e) \times (n \text{ rem } B^e)$. For some multiplication methods, such as the classical $O(N^2)$ algorithm or the asymptotically faster Karatsuba algorithm, computing the lower digits of this product will be faster than computing the full product by a constant factor. For other methods, there will be no benefit beyond that provided by having the shorter multiplicands $v \text{ rem } B^e$ and $n \text{ rem } B^e$.

The value of e will be determined by the precisions $\mathrm{prec}\, v = k+1$ and $\mathrm{prec}\, w = t+1$, the number ℓ of known correct places in w and the required number g of guard digits, as

$$e \le k + t - \ell + g. \tag{19}$$

6 INTEGER ALGORITHM

Algorithm 1 presents the results of Section 5 in computational form. The initial stages of Shinv (lines 2-10) guarantee that the base-B is sufficiently large and that certain easy cases are handled, so $B < v < B^h/2$. The next section (lines 11-15) forms an initial value for the iteration that guarantees fast convergence. This initial value may have sufficiently many correct digits to be shifted to the required length and returned directly. otherwise, this initial value is refined in an iteration.

Three variants of refinement are given, incorporating the results of Theorems 10 and 11 in stages, with Refine3 being the method to use in practice. In each case w is the value that converges to B^h quo v and ℓ is the number of leading correct base-B digits. The remainder of the procedure refines the result iteratively using one of the Refine methods. Each of these makes use of the Step procedure (lines 43-44), which computes the function given in equation (13) with the additional parameters m, ℓ and B. Here, ℓ is the number of correct leading digits of w. The parameter m gives the number of additional digits needed, since on the last iteration it will not be necessary to double the number. This is passed as a parameter so

that only the instances of w in Step are shifted, giving smaller products. All of the Refine methods make use of PowDiff to compute $B^h - v \cdot w$ as described in Section 5.4 and detailed in Algorithm 2.

Refine1 gives a naïve iteration where all computations are performed at the full length of the final result. By shifting one digit, the iteration avoids terminating at the $\lfloor B^h/v \rfloor - 1$ fixed point. No guard digits are needed in the intermediate computation.

Refine2 adjusts the iteration so that only the accurate digits of the intermediate results are computed, as described in Section 5.2. Two guard digits are required since Theorem 10 shows the short iterate can differ from the truncated full length value by up to B.

Refine3 additionally uses short divisor prefixes, when possible. Refine3 is the same as Refine2 when $s = 0$. Two guard digits are required since both the short iterate computation and the use of a divisor prefix can together give a value that is upto $2B$ off from the truncated full length iterate value. The variable s is as given by equation (17), taking into account the guard digits.

Using PowDiff is most beneficial in Refine1, while the use of short iterates and divisor prefixes in Refine2 and Refine3 already provide a part of this benefit. A low level implementation would access the base-B digits directly and pre-allocate and re-use a storage region sufficient to hold the largest intermediate results.

The time complexity to compute Shinv depends on the choice of Refine and on the multiplication method used for Mult and MultMod. In the following analysis, we assume that the time to compute MultMod(a, b, d, B) is of the same order as $M(N)$ where $N = \max(\min(\log a, d), \min(\log b, d)))$, *i.e.* that of computing Mult$(a \text{ rem } B^d,\ b \text{ rem } B^d) \text{ rem } B^d$. For FFT multiplication these times will be the same, for other methods they may differ by a constant factor.

In all cases, Shinv performs $\lceil \log_2(h-k) \rceil$ iterations, each having two multiplications. When using Refine1, if $k < h/2$, one multiplication will be of arguments of length between $h - k$ and $h/2$ and the other of length between h and $h - k$, *i.e.* of time $O(M(h))$ and $O(M(h - k))$. If $k > h/2$, one multiplication will be of arguments of length k and the other of arguments of length between h and k, *i.e.* of time $O(M(h))$ and $M(k)$. Together these give $T(h, k) \in O\big(\log(h - k)(M(h) + M(|h/2 - k|))\big)$. Here we have ignored additive constants.

When using Refine2 or Refine3 only the necessary prefixes are computed. When using Refine2, if $k < h/2$, one multiplication will be of length k and the other of length $\max(k, 2^{i-1}) < h/2$. If $k > h/2$ both multiplications will be of arguments of length k.

With Refine3, at iteration i one multiplication will be of arguments of length 2^i. If $k > h/2$, the second multiplication will be of arguments also of this length. If $k < h/2$, the second multiplication will be of arguments of length $\min(2^i, k)$. Again, we have ignored additive constants. In all cases, letting $h = k + N$,

$$T(k + N, k) \in O\left(\sum_{i=1}^{\log N} M(2^i)\right).$$

This results in time complexity $O(M(N))$ for the theoretical $M(N) \in O(N \log N)$, for Schönhage-Strassen $M(N) \in O(N \log N \log\log N)$ and for practical $M(N) \in O(N^p)$, $p > 0$.

7 POLYNOMIAL AND GENERIC ALGORITHMS

Polynomial Algorithm

It is straightforward to adapt Algorithm 1 to univariate polynomials over a field. The iteration step function is given by equation (3). Algorithm 3 shows the details. Polynomial versions of the three Refine methods are shown, but it is Refine3 that should be used.

As usual, the polynomial algorithm is simpler than the integer version. Often iterative methods for polynomials start with a monomial. Here we start with two terms to simplify the termination condition and the generic algorithm. Once an initial value is determined, the only changes from Algorithm 1 relate to the consequences of integer arithmetic having carries. There is no longer any need for guard digits, nor being one short of doubling precision, nor change of base if B is too small. Although it is not strictly necessary, we retain the PowDiff operation. This emphasizes the parallel between the integer and polynomial algorithms, and may provide efficiencies when polynomials are stored densely.

Generic Algorithm

The benefit of using the whole shifted inverse is that the arithmetic remains in the original domain. This allows the iterative algorithm to be defined generically on a domain D with suitable shift, as shown in Algorithm 4.

The generic versions of Refine1, Refine2 and Refine3 may be used in place of the function of the same name in Algorithm 1 or 3. It would indeed have been possible to present this generic algorithm first, and show the integer and polynomial cases as specializations, but that would have been less clear.

While the operations Mult and MultMod are mathematically simple, they will typically be implemented by methods provided as procedural parameters. When there are carries, it is not possible to double the number of correct places at each step and the variable d gives the shortfall. When the arithmetic has no carries, no guard digits are required. The complexity analysis of the integer algorithm carries over directly to the polynomial and generic versions. An implementation should be able to provide shift as an $O(1)$ operation.

The domain D need not be commutative. It must, however, have a suitable whole shift operation. The shift must be with respect to a central element $b \in D$ (*i.e.*, $bd = db$ for all $d \in D$) and subset $S \subseteq D$ such that every element $d \in D$ can be expressed uniquely as $d = \sum_{i=0}^{k} d_i b^i$ with $d_i \in S$. In this case, the generic versions of Refine1, Refine2 and Refine3 all apply. An example of such a ring would be the matrix polynomials $D = F^{n\times n}[x]$ with central element $b = x$. For u, v in such D, there exist left and right quotients and remainders $q_L, r_L, q_R, r_R \in D$ such that

$$u = v\, q_L + r_L = q_R\, v + r_R$$

with $r = 0$ or $N(r) < N(v)$ for $r \in \{r_L, r_R\}$ and Euclidean norm function N being the degree in b. The operations shift and shinv are well-defined and may be used to compute the quotients as

$$q_L = \text{shift}_{-h}(\text{shinv}_h\, v \cdot u) \qquad q_R = \text{shift}_{-h}(u \cdot \text{shinv}_h\, v).$$

For polynomials where the variable does not commute with the coefficients, *e.g.* $R[y][\partial_y]$, this method applies only in a limited way. The non-commutative case is discussed further in [19].

8 CONCLUSIONS

We have shown how to compute whole shifted inverses and quotients for integers and univariate polynomials with the same order of complexity as multiplication and requiring only domain-preserving ring operations and shifts. The algorithms are practical and can be used to modularize software libraries. Several results pertaining to the fixed points and convergence of the integer iteration are proven to establish the soundness and efficiency of the algorithm. We have presented the iterative algorithm generically for domains, not necessarily commutative, endowed with a suitable whole shift operation.

ACKNOWLEDGMENTS

We thank Reviewer 3 for a careful reading and colleagues for helpful comments. This work was supported in part by a grant from the University of Waterloo.

REFERENCES

[1] P. Afshani, C.B. Freksen, L. Kamma, and K.G. Larsen. 2019. Lower Bounds for Multiplication via Network Coding. In *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, Vol. 132. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 10:1–10:12.

[2] Alfred V. Aho, John E Hopcroft, and Jeffrey D. Ullman. 1974. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass.

[3] Paul Barrett. 1987. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *Advances in Cryptology — CRYPTO'86.* Springer, New York, 311–323.

[4] Daniel J. Bernstein. 2008. Fast Multiplication and Its Applications. In *Algorithmic Number Theory*, J.P. Buhler and P. Stevehagen (Eds.). Cambridge University Press, Cambridge, 325–384.

[5] Christoph Burnikel and Joachim Ziegler. 1998. *Fast Recursive Division.* Technical Report MPI-I-98-1-022. Max-Planck-Institut f ur Informatik, Saarbrcken, Germany.

[6] Stephen A. Cook. 1966. *On the Minimum Computation Time of Functions.* Ph. D. Dissertation. Harvard University.

[7] Pacal Giorgi, Bruno Grenet, and Daniel S. Roche. 2020. Fast in-place algorithms for polynomial operations: division, evaluation and interpolation. In *Proc. 2020 International Symposium on Symbolic and Algebraic Computation (ISSAC 2020.* ACM, New York, 210–217.

[8] GMP Development Team. 2020. *The GNU Multiple Precision Arithmetic Library (version 6.2.1).* Free Software Foundation. https://gmplib.org

[9] David Harvey and Joris van der Hoeven. 2021. Integer Multiplication in Time $n \log n$. *Annals of Mathematics* 193 (2021), 563–617. Issue 2.

[10] William Hasenplaugh, Gunnar Gaubatz, and Vinodh Gopal. 2007. Fast Modular Reduction. In *18th IEEE Symposium on Computer Arithmetic (ARITH '07).* IEEE, Washington DC, 225–229. https://doi.org/10.1109/ARITH.2007.18

[11] Markus Hitz and Erich Kaltofen. 1995. Integer division in residue number systems. *IEEE Trans. Comput.* 44, 8 (1995), 983–989.

[12] Tudor Jebelean. 1997. Practical Division with Karatsuba Complexity. In *Proc. 1997 International Symposium on Symbolic and Algebraic Computation (ISSAC 1997.* ACM, New York, 339–341.

[13] Anatoly Karatsuba and Ofman Yu. 1962. Multiplication of Many-Digital Numbers by Automatic Computers. *Proceedings of the USSR Academy of Sciences* 145 (1962), 293–294. Translation in the academic journal Physics-Doklady, 7 (1963), pp. 595–596.

[14] Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms* (third ed.). Addison-Wesley, Boston.

[15] Donald E. Knuth. 2022. *The Art of Computer Programming, Volume 4b: Combinatorial Algorithms, Part 2.* Addison-Wesley, Boston.

[16] Robert T. Moenck and Allan B. Borodin. 1972. Fast Modular Transforms via Division. In *Proc. 13th Annual Symposium on Switching and Automata Theory (SWAT 1972).* IEEE, New York, 90–96.

[17] Arnold Schönhage and Volker Strassen. 1971. Schnelle Multiplikation großer Zahlen. *Computing* 7 (1971), 281––292.

[18] Joachim von zur Gathen and Jürgen Gerhard. 2013. *Modern Computer Algebra* (third ed.). Cambridge University Press, Cambridge.

[19] Stephen M. Watt. 2023. Efficient Quotients of Non-Commutative Polynomials. In *25th International Workshop on Computer Algebra in Scientific Computing (CASC 2023).* Springer LNCS (to appear), 23 pages.

Algorithm 1 SHINV(v, h, B) in $\mathbb{Z}$

Input: $v, h, B \in \mathbb{Z}_{>0}$, $B^k \le v < B^{k+1}$
Output: $\text{shinv}_h\, v$ ▷ *All* shift*s* shinv*s are with respect to B*
Uses: MULT, a multiplication method
POWDIFF, to compute $B^h - vw$ (Algorithm 2)
REFINE one of REFINE1, REFINE2 or REFINE3

function SHINV(v, h, B)
  ▷ *Group digits if base is small.*
  if $B < 16$ **then**
    $p \leftarrow \max(6 - B, 2)$
    return $\text{shift}_{h \bmod p - p}$ SHINV$(v, h \text{ quo } p + 1, B^p)$
  ▷ *Special cases guarantee $B < v \le B^h/2$.*
  if $v < B$ **then return** B^h quo v ▷ *Divide by 1 digit*
  if $v > B^h$ **then return** 0
  if $2v > B^h$ **then return** 1
  if $v = B^k$ **then return** B^{h-k}
  ▷ *Form initial approximation, returning it if sufficient.*
  $\ell \leftarrow \min(k, 2)$
  $V \leftarrow \sum_{i=0}^{\ell} v_{k-\ell+i} B^i$
  $w \leftarrow (B^{2\ell} - V)$ quo $V + 1$ ▷ *Divide 4 digits by 2 digits*
  if $h - k \le \ell$ **then return** $\text{shift}_{h-k-\ell}(w)$
  ▷ *Refine iteratively using one of the methods below.*
  return REFINE(v, h, k, w, ℓ)

function REFINE1(v, h, k, w, ℓ)
  $g \leftarrow 1$
  $h \leftarrow h + g$
  $w \leftarrow \text{shift}_{h-k-\ell}(w)$ ▷ *Scale initial value to full length*
  while $h - k > \ell$ **do**
    $w \leftarrow$ STEP$(h, v, w, 0, \ell, 0)$
    $\ell \leftarrow \min(2\ell - 1,\ h - k)$ ▷ *Number of correct digits*
  return $\text{shift}_{-g}(w)$

function REFINE2(v, h, k, w, ℓ)
  $g \leftarrow 2$ ▷ *2 guard digits*
  $w \leftarrow \text{shift}_g\, w$
  while $h - k > \ell$ **do**
    $m \leftarrow \min(h - k + 1 - \ell,\ \ell)$ ▷ *How much to grow*
    $w \leftarrow \text{shift}_{-1}$ STEP$\big(k + \ell + m + g,\ v,\ w\ ,\ \ell,\ g\big)$
    $\ell \leftarrow \ell + m - 1$
  return $\text{shift}_{-g}(w)$

function REFINE3(v, h, k, w, ℓ)
  $g \leftarrow 2$ ▷ *2 guard digits*
  $w \leftarrow \text{shift}_g\, w$
  while $h - k > \ell$ **do**
    $m \leftarrow \min(h - k + 1 - \ell,\ \ell)$
    $s \leftarrow \max(0,\ k - 2\ell + 1 - g)$ ▷ *How to scale v*
    $w \leftarrow \text{shift}_{-1}$ STEP$\big(k+\ell+m-s+g,\ \text{shift}_{-s}\ v,\ w,\ m,\ \ell,\ g\big)$
    $\ell \leftarrow \ell + m - 1$
  return $\text{shift}_{-g}(w)$

function STEP(h, v, w, m, ℓ, g)
  $\text{shift}_m\, w + \text{shift}_{2m-h}$ MULT$\big(w, \text{POWDIFF}(v, w, h-m, \ell-g, B)\big)$

Algorithm 2 PowDiff(v, w, h, ℓ, B) in $\mathbb{Z}$

Input: $v, w, h, \ell, B \in \mathbb{Z}_{>0}$ where $\text{prec}\left| w - \text{shinv}_h\, v \right| \leq \text{prec}\, w - \ell$
Output: $B^h - v \cdot w$
Uses: Mult$(a, b) = a \cdot b$,
MultMod$(a, b, d, B) = (a \cdot b) \text{ rem } B^d$

```
1: function PowDiff(v, w, h, ℓ, B)
2:     L ← prec_B v + prec_B w − ℓ + 1
3:     if v = 0 ∨ w = 0 ∨ L ≥ h then return B^h − Mult(v, w)
4:     else
5:         P ← MultMod(v, w, L, B)
6:         if P = 0 then return 0
7:         else if P_{L−1} = 0 then return −P
8:         else return B^L − P
```

Algorithm 3 Shinv(v, h) in $F[x]$

Input: $v \in F[x], h \in \mathbb{Z}_{>0}$ where $k = \text{prec}\, v - 1$ and F a field
Output: $\text{shinv}_h\, v$ ▷ *All* shift*s* shinv*s are with respect to x*
Uses: Mult$(a, b) = a \cdot b$,
MultMod$(a, b, d) = (a \cdot b) \text{ rem } x^d$,
Refine one of Refine1, Refine2 or Refine3

```
1: function Shinv(v, h)
2:     ▷ Special cases. Afterward 0 < k < h.
3:     if k > h then return 0
4:     if k = 0 ∨ k = h ∨ v = v_k x^k then return x^{h−k}/v_k
5:     ▷ Form initial approximation.
6:     w ← x/v_k − v_{k−1}/v_k^2; ℓ ← 2
7:     ▷ Refine iteratively using one of the methods below.
8:     return Refine(v, h, k, w, ℓ)
9: function Refine1(v, h, k, w, ℓ)
10:    w ← shift_{h−k−ℓ}(w)          ▷ Scale initial value to full length
11:    while h − k + 1 > ℓ do
12:        w ← Step(h, v, w, 0, ℓ)
13:        ℓ ← min(2ℓ, h − k + 1)          ▷ Number of correct digits
14:    return w
15: function Refine2(v, h, k, w, ℓ)
16:    while h − k + 1 > ℓ do
17:        m ← min(h − k + 1 − ℓ, ℓ)          ▷ How much to grow
18:        w ← Step(k + ℓ + m − 1, v, w, m, ℓ)
19:        ℓ ← ℓ + m
20:    return w
21: function Refine3(v, h, k, w, ℓ)
22:    while h − k + 1 > ℓ do
23:        m ← min(h − k + 1 − ℓ, ℓ)
24:        s ← max(0, k − 2ℓ + 1)          ▷ How to scale v
25:        w ← Step(k + ℓ + m − 1 − s, shift_{−s} v, w, m, ℓ)
26:        ℓ ← ℓ + m
27:    return w
28: function Step(h, v, w, m, ℓ)
29:    shift_m w + shift_{2m−h} Mult(w, PowDiff(v, w, h − m, ℓ))
30: ▷ Compute x^h − v · w
31: function PowDiff(v, w, h, ℓ)
32:    L ← prec v + prec w − ℓ
33:    if v = 0 ∨ w = 0 ∨ L ≥ h then return x^h − Mult(v, w)
34:    else return −MultMod(v, w, L)
```

Algorithm 4 Generic Refines, Step and PowDiff

Certain operations are required on D. On $\mathbb{Z}$ in base-B, these are

$$\begin{aligned} \text{shift}_n\, u &= \lfloor u B^n \rfloor \\ \text{coeff}(u, i) &= u_i \\ \text{HasCarries} &= \text{true} \\ \text{Mult}(a, b) &= ab \\ \text{MultMod}(a, b, n) &= ab \text{ rem } B^n. \end{aligned}$$

On $F[x]$, for F a field, these are

$$\begin{aligned} \text{shift}_n\, u &= u \cdot x^n \text{ if } n \geq 0,\ u \text{ quo } x^{-n} \text{ if } n < 0 \\ \text{coeff}(u, i) &= u[i] \\ \text{HasCarries} &= \text{false} \\ \text{Mult}(a, b) &= ab \\ \text{MultMod}(a, b, n) &= ab \text{ rem } x^n. \end{aligned}$$

```
1: ▷ Below, g is no. guard places and d is the prec. doubling shortfall.
2: function D.Refine1(v, h, k, w, ℓ)
3:     if D.HasCarries then g ← 1; d ← 1 else g ← 0; d ← 0
4:     h ← h + g
5:     w ← D.shift_{h−k−ℓ+1−g}(w) ▷ Scale init. value to full length
6:     while h − k + 1 − d > ℓ do
7:         w ← D.Step(h, v, w, 0, ℓ)
8:         ℓ ← min(2ℓ − d, h − k + 1)    ▷ Number of correct digits
9:     return D.shift_{−g} w
10: function D.Refine2(v, h, k, w, ℓ)
11:    if D.HasCarries then g ← 2; d ← 1 else g ← 0; d ← 0
12:    w ← D.shift_g w
13:    while h − k + 1 − d > ℓ do
14:        m ← min(h − k + 1 − ℓ, ℓ)          ▷ How much to grow
15:        w ← D.shift_{−d} D.Step(k+ℓ+m+d−1+g, v, w, m, ℓ−g)
16:        ℓ ← ℓ + m − d
17:    return D.shift_{−g} w
18: function D.Refine3(v, h, k, w, ℓ)
19:    if D.HasCarries then g ← 2; d ← 1 else g ← 0; d ← 0
20:    w ← D.shift_g w
21:    while h − k + 1 − d > ℓ do
22:        m ← min(h − k + 1 − ℓ, ℓ)
23:        s ← max(0, k − 2ℓ + 1 − g)
24:        t ← k + ℓ + m − s + d − 1 + g
25:        w ← D.shift_{−d} (D.Step(t, D.shift_{−s} v, w, m, ℓ − g))
26:        ℓ ← ℓ + m − d
27:    return D.shift_{−g} w
28: function D.Step(h, v, w, m, ℓ)
29:    return D.shift_m w +
30:        D.shift_{2m−h} D.Mult(w, D.PowDiff(v, w, h − m, ℓ))
31: function D.PowDiff(v, w, h, ℓ)
32:    c ← if D.HasCarries then 1 else 0
33:    L ← D.prec v + D.prec w − ℓ + c          ▷ c for coeff to peek
34:    if v = 0 ∨ w = 0 ∨ L ≥ h then
35:        return D.shift_h 1 − D.Mult(v, w)
36:    else
37:        P ← D.MultMod(v, w, L)
38:        if D.HasCarries ∧ D.coeff(P, L − 1) ≠ 0 then
39:            return D.shift_L 1 − P
40:        else return −P
```