Efficient Quotients of Non-Commutative Polynomials

Stephen M. Watt

Cheriton School of Computer Science, University of Waterloo
https://cs.uwaterloo.ca/~smwatt
smwatt@uwaterloo.ca

Abstract. It is shown how to compute quotients efficiently in non-commutative univariate polynomial rings. This extends earlier work where efficient generic quotients were studied with a primary focus on commutative domains. Fast algorithms are given for left and right quotients of polynomials where the variable commutes with coefficients. These algorithms are based on the concept of the "whole shifted inverse", which is a specialized quotient where the dividend is a power of the polynomial variable. It is also shown that when the variable does not commute with coefficients, that is for skew polynomials, left and right whole shifted inverses are defined and may be used to compute right and left quotients. In this case their computation is not asymptotically fast, but once obtained, they may be used to compute multiple quotients, each with one multiplication. Examples are shown of polynomials with matrix coefficients, differential operators and difference operators. In addition, a proof-of-concept generic Maple implementations is given.

1 Introduction

In symbolic mathematical computation it is important to have efficient algorithms for the fundamental arithmetic operations of addition, multiplication and division. While linear time algorithms for additive operations are usually straightforward, considerable attention has been devoted to find efficient methods to compute products and quotients of integers, polynomials with integer or finite field coefficients and matrices with elements from a ring. For these, both practically efficient algorithms and theoretically important bounds are well known.

For integer and polynomial division, efficient algorithms based on Newton iteration allow the computation of quotients in time proportional to multiplication. Until recently, these algorithms left the original domain to perform arithmetic in related domains. For integers, this involved computing an approximation to the inverse of the divisor in extended precision approximate arithmetic or in a

residue ring, and for polynomials it involved computing the inverse of the reverse of the divisor polynomial in ideal-adic arithmetic.

We have recently shown how these quotients may be computed without leaving the original domain, and we have extended this to a generic domain-preserving algorithm for rings with a suitable whole shift operation [10]. For integers the whole shift multiplies by a power of the representation base and for polynomials it multiplies by a power of the variable, in both cases discarding terms with negative powers. The previous paper developed the concept of the whole shifted inverse and used it to compute quotients efficiently. Non-commutative domains were mentioned only briefly.

The present article expands on how these methods may be used to compute quotients of non-commutative polynomials. In particular, it is shown that

- the whole shifted inverse is well-defined on non-commutative polynomial rings $R[x]$,
- its computation is efficient,
- they may be used to compute left or right quotients in $R[x]$, each with one multiplication,
- left and right whole shifted inverses may be defined on skew polynomials $R[x;\sigma,\delta]$, and
- they may be used to compute the right and left quotients in $R[x;\sigma,\delta]$, each with one multiplication.

The remainder of this article is organized as follows. Section 2 presents some basic background, including notation, the definition of division in a non-commutative context, and the Newton-Schulz iteration. Section 3 considers division of non-commutative polynomials in $R[x]$, showing $O(n^2)$ algorithms for classical division and for pseudodivision. It recalls the notion of the whole shifted inverse, proves it is well-defined on non-commutative $R[x]$ and shows that it can be used to compute left and right quotients in this setting. Section 4 recapitulates the generic algorithms from [10] that use a modified Newton iteration to compute the whole shifted inverse. It also explains why it applies when polynomial coefficients are non-commutative. Section 5 gives an example of these algorithms applied to polynomial matrices. Section 6 extends the discussion to skew polynomials $R[x;\sigma,\delta]$, defining left and right whole shifted inverse, and showing how they may be used. Section 7 gives linear ordinary differential and difference operators as examples, before concluding remarks in Section 8.

2 Background

2.1 Notation

We adopt the following notation:

$\mathrm{prec}_B\, u$	number of base-B digits of an integer u, $\lfloor \log_B \lvert u \rvert \rfloor + 1$
$\mathrm{prec}_x\, p$	number of coefficients of a polynomial p, $\mathrm{degree}_x\, p + 1$
$u \operatorname{quo} v,\ u \operatorname{rem} v$	quotient and remainder (see below)
$u \operatorname{\mathbf{x}quo} v,\ u \operatorname{\mathbf{x}rem} v$	left and right (pseudo)quotient and remainder, $\mathbf{x} \in \{\mathrm{l}, \mathrm{lp}, \mathrm{r}, \mathrm{pr}\}$
$\mathrm{shift}_n\, v$, $\mathrm{shinv}_n\, v$	whole shift and whole shifted inverse (see below)
$R[x; \sigma, \delta]$, $R[x, \delta]$	skew polynomials (see Section 6)
${}_i u$, u_i	coefficient of skew polynomial u with variable powers on the left, right.
$\mathbf{x}\mathrm{shift}_n v$, $\mathbf{x}\mathrm{shinv}_n v$	left and right whole shift and shifted inverse, $\mathbf{x} \in \{\mathrm{l}, \mathrm{r}\}$ (see Section 6)
$X_{(i)}$	value of X at i^{th} iteration

The "prec" notation, standing for "precision", means the number of base-B digits or polynomial coefficients. It is similar to that of [4], where it is used to present certain algorithms generically for integers and polynomials. In particular, if we take integers to be represented in base-B, *i.e.* for any integer $u \neq 0$ there is $h = \mathrm{prec}_B(u) - 1$, such that

$$u = \sum_{i=0}^{h} u_i B^i, \quad u_i \in \mathbb{Z},\ 0 \leq u_i < B,\ u_h \neq 0, \tag{1}$$

then integers base-B behave similarly to univariate polynomials with coefficients u_i, but with carries complicating matters.

2.2 Division

The notion of integer quotients and remainders can be extended to more general rings. For a Euclidean domain D with valuation $N : D \to \mathbb{Z}_{\geq 0}$, such that for any $u, v \in D, v \neq 0$, there exist $q, r \in D$ such that

$$u = qv + r, \qquad r = 0 \text{ or } N(r) < N(v).$$

The value q is a *quotient* of u and v and r is a *remainder* of dividing u by v and we write

$$q = u \operatorname{quo} v \qquad r = u \operatorname{rem} v$$

when these are unique. When both the quotient and remainder are required, we write $u \operatorname{div} v = (u \operatorname{quo} v, u \operatorname{rem} v)$. When D is a non-commutative ring with a valuation N, there *may* exist left and right quotients such that

$$\begin{aligned} u &= v\, q_{\mathrm{L}} + r_{\mathrm{L}}, & r_{\mathrm{L}} &= 0 \text{ or } N(r_{\mathrm{L}}) < N(v) \\ u &= q_{\mathrm{R}}\, v + r_{\mathrm{R}}, & r_{\mathrm{R}} &= 0 \text{ or } N(r_{\mathrm{R}}) < N(v). \end{aligned} \tag{2}$$

When these exist and are unique, we write

$$q_{\mathrm{L}} = u \operatorname{lquo} v \qquad r_{\mathrm{L}} = u \operatorname{lrem} v \qquad q_{\mathrm{R}} = u \operatorname{rquo} v \qquad r_{\mathrm{R}} = u \operatorname{rrem} v.$$

For certain non-commutative rings with a distance measure $\|\cdot\|$, a sequence of approximations to the inverse of A may be computed via the Newton-Schulz iteration [7]

$$X_{(i+1)} = X_{(i)} + X_{(i)}(1 - AX_{(i)}) \tag{3}$$

where 1 denotes the multiplicative identity of the ring. There are several ways to arrange this expression, but the form above emphasizes that as $X_{(i)}$ approaches A^{-1}, the product $X_{(i)}(1 - AX_{(i)})$ approaches 0. For $\mathbb{C}^{n\times n}$ matrices, a suitable initial value is $X_{(0)} = A^\dagger/(n \operatorname{Tr}(AA^\dagger))$, where $A^\dagger$ is the Hermitian transpose.

2.3 Whole Shift and Whole Shifted Inverse

In previous work [10] we studied the problem of efficient domain-preserving computation of quotients and remainders for integers and polynomials, then generalized these results to a generic setting. To this end, we defined the notions of the *whole shift* and *whole shifted inverse* with attention to commutative domains. We recapitulate these definitions and two results relevant to the present article.

Definition 1 (Whole n-shift in $R[x]$) *Given a polynomial $u = \sum_{i=0}^{h} u_i x^i \in R[x]$, with R a ring and $n \in \mathbb{Z}$, the* whole n-shift of u with respect to x *is*

$$\operatorname{shift}_{n,x} u = \sum_{i+n\geq 0} u_i x^{i+n}. \tag{4}$$

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

Definition 2 (Whole n-shifted inverse in $F[x]$) *Given $n \in \mathbb{Z}_{\geq 0}$ and $v \in F[x]$, F a field, the* whole n-shifted inverse of v with respect to x *is*

$$\operatorname{shinv}_{n,x} v = x^n \operatorname{quo} v. \tag{5}$$

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

Theorem 1 *Given two polynomials $u, v \in F[x]$, F a field, and $0 \leq \operatorname{degree} u \leq h$,*

$$u \operatorname{quo} v = \operatorname{shift}_{-h}(u \cdot \operatorname{shinv}_h v). \tag{6}$$

For classical and Karatsuba multiplication it is more efficient to compute just the top part of the product in (6), omitting the lower h terms, instead of shifting:

$$\operatorname{shift}_{-h}(u \cdot \operatorname{shinv}_h v) = \textsc{MultQuo}(u, \operatorname{shinv}_h v, h),$$

with $\textsc{MultQuo}(a, b, n) = ab \operatorname{quo} x^n$ computing only $\operatorname{degree} a + \operatorname{degree} b - n + 1$ terms. For multiplication methods where computing only the top part of the product gives no saving, some improvement is obtained using

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

Algorithm 1 Classical division for non-commutative $R[x]$ with invertible v_k

▷ *Compute* $q = \sum_{i=0}^{h-k} q_i x^i$ *and* $r = \sum_{i=0}^{k-1} r_i x^i$ *such that* $u = q \times_\pi v + r$.
function DIV $(u = \sum_{i=0}^{h} u_i x^i \in R[x], v = \sum_{i=0}^{k} v_i x^i \in R[x], \pi \in S_2)$
    $v^* \leftarrow \text{inv } v_k$
    $q \leftarrow 0$
    $r \leftarrow u$
    for $i \leftarrow h - k$ to 0 by -1 **do**
        $t \leftarrow (r_{i+k} \times_\pi v^*)\, x^i$
        $q \leftarrow q + t$
        $r \leftarrow r - t \times_\pi v$
    return (q, r)
▷ *Left division:* $(q_{\text{L}}, r_{\text{L}}) \leftarrow \text{LDIV}(u, v) \Rightarrow u = v \times q_{\text{L}} + r_{\text{L}}$
$\text{LDIV}(u, v) \mapsto \text{DIV}\big(u, v, (2\,1)\big)$
▷ *Right division:* $(q_{\text{R}}, r_{\text{R}}) \leftarrow \text{RDIV}(u, v) \Rightarrow u = q_{\text{R}} \times v + r_{\text{R}}$
$\text{RDIV}(u, v) \mapsto \text{DIV}\big(u, v, (1\,2)\big)$

Theorem 2 *Given* $v \in F[x]$*, with* F *a field and* $h > \operatorname{degree} v = k$ *and suitable starting value* $w_{(0)}$*, the sequence of iterates*

$$w_{(i+1)} = w_{(i)} + \operatorname{shift}_{-h}\left(w_{(i)}(\operatorname{shift}_h 1 - v w_{(i)})\right)$$

converges to $\operatorname{shinv}_h v$ *in* $\lceil \log_2(h-k) \rceil$ *steps.*

A suitable starting value for $w_{(0)}$ is given by SHINV0 in Section 4.

3 Division in Non-Commutative $R[x]$

We now lay out how to use shift and shinv to compute quotients for polynomials with non-commutative coefficients. First we show classical algorithms to compute left and right quotients in $R[x]$. We then prove two theorems, one showing that $x^n \operatorname{lquo} v = x^n \operatorname{rquo} v$ in this setting, making the whole shifted inverse well defined, and another showing that it may be used to compute left and right quotients.

3.1 Definitions and Classical Algorithms

Let u and v be two polynomials in $R[x]$ with Euclidean norm being the polynomial degree. The left and right quotients and remainders are defined as in (2). Left and right quotients will exist provided that v_k is invertible in R and they may be computed by Algorithm 1. In the presentation of the algorithm, π denotes a permutation on two elements so is either the identity or a transposition. The notation $\times_\pi$ is a shorthand for $\times \circ \pi$ so $a \times_\pi b = a \times b$ when π is the identity and $a \times_\pi b = b \times a$ when π is a transposition.

Algorithm 2 Non-commutative polynomial pseudodivision

▷ *Compute* $q = \sum_{i=0}^{h-k} q_i x^i$ *and* $r = \sum_{i=0}^{k-1} r_i x^i$ *such that* $v_k^{h-k+1} u = q \times_\pi v + r$. *Requires* $v \times v_k = v_k \times v$.

function PDIV $(u = \sum_{i=0}^{h} u_i x^i \in R[x], v = \sum_{i=0}^{k} v_i x^i \in R[x], \pi \in S_2)$
    $q \leftarrow 0$
    $r \leftarrow u$
    for $i \leftarrow h - k$ to 0 by -1 **do**
        $t \leftarrow u_{i+k}\, x^i$
        $q \leftarrow q + t \times_\pi v_k^i$
        $r \leftarrow r \times_\pi v_k - t \times_\pi v$
    return (q, r)

▷ *Left pseudodivision:* $(q_{\text{L}}, r_{\text{L}}) \leftarrow \text{LPDIV}(u, v) \Rightarrow v_k^{h-k+1} u = v \times q_{\text{L}} + r_{\text{L}}$
$\text{LPDIV}(u, v) \mapsto \text{PDIV}\big(u, v, (2\,1)\big)$

▷ *Right pseudodivision:* $(q_{\text{R}}, r_{\text{R}}) \leftarrow \text{RPDIV}(u, v) \Rightarrow v_k^{h-k+1} u = q_{\text{R}} \times v + r_{\text{R}}$
$\text{RPDIV}(u, v) \mapsto \text{PDIV}\big(u, v, (1\,2)\big)$

There are some circumstances where quotients or related quantities may be computed even if v_k is not invertible. When R is an integral domain, quotients may be computed as usual in $K[x]$ with K being the quotient field of R. Alternatively, when R is non-commutative but v_k commutes with v, it is possible to compute *pseudoquotients* and *pseudoremainders* satisfying

$$\begin{aligned} m\,u &= v\,q_{\text{L}} + r_{\text{L}}, & &\text{degree}\, r_{\text{L}} < \text{degree}\, v \\ u\,m &= q_{\text{R}}\,v + r_{\text{R}}, & &\text{degree}\, r_{\text{R}} < \text{degree}\, v \\ m &= v_k^{h-k+1}, \end{aligned}$$

as shown in Algorithm 2. In this case, we write

$$\begin{aligned} q_{\text{L}} &= u\,\text{lpquo}\,v & r_{\text{L}} &= \text{lprem}\,v \\ q_{\text{R}} &= u\,\text{rpquo}\,v & r_{\text{L}} &= \text{rprem}\,v. \end{aligned}$$

Requiring v_k to commute with v is quite restrictive, however, so we focus our attention to situations where the inverse of v_k exists.

3.2 Whole Shift and Whole Shifted Inverse in $R[x]$

We now examine the notions of the whole shift and whole shifted inverse for $R[x]$ with non-commutative R. First consider the whole shift. Since x commutes with all values in $R[x]$, we may without ambiguity take, for $u = \sum_{i=0}^{h} u_i x^i$ and $n \in \mathbb{Z}$,

$$\text{shift}_n\, u \;=\; \sum_{i+n\geq 0} x^n (u_i x^i) \;=\; \sum_{i+n\geq 0} (u_i x^i) x^n. \tag{7}$$

That is, the fact that $R[x]$ is non-commutative does not lead to left and right variants of the whole shift.

We state two simple theorems with obvious proofs:

Theorem 3 *Let $w \in R[x]$. Then, for all $n \in \mathbb{Z}_{\geq 0}$, $\operatorname{shift}_{-n} \operatorname{shift}_n w = w$.*

Theorem 4 *Let $u, v \in R[x]$ with $\operatorname{degree} u = h$ and $\operatorname{degree} v = k$. Then, for $m \in \mathbb{Z}$,*

$$\operatorname{shift}_{-k-m}(u \times v) = \operatorname{shift}_{-k}(\operatorname{shift}_{-m}(u) \times v)$$
$$\operatorname{shift}_{-h-m}(u \times v) = \operatorname{shift}_{-h}(u \times \operatorname{shift}_{-m}(v)).$$

We now come to the main point of this section and show shinv is well-defined when R is non-commutative.

Theorem 5 (Whole shifted inverse for non-commutative $R[x]$)
Let $v = \sum_{i=0}^{k} v_i x^i \in R[x]$, with R a non-commutative ring and v_k invertible in R. Then, for $h \in \mathbb{Z}_{\geq 0}$,

$$x^h \operatorname{lquo} v = x^h \operatorname{rquo} v.$$

Proof. Let $q_{\mathrm{L}} = x^h \operatorname{lquo} v$ and $q_{\mathrm{R}} = x^h \operatorname{rquo} v$. If $h < k$, then $q_{\mathrm{L}} = q_{\mathrm{R}} = 0$. Otherwise, both q_{L} and q_{R} have degree $h - k \geq 0$ so

$$v_k\, q_{\mathrm{L}h-k} = 1 \qquad q_{\mathrm{R}h-k}\, v_k = 1 \tag{8}$$

$$\sum_{j=M}^{k} v_j\, q_{\mathrm{L}i+k-j} = 0 \qquad \sum_{j=M}^{k} q_{\mathrm{R}i+k-j}\, v_j = 0, \quad 0 \leq i < h-k, \tag{9}$$

where $M = \max(0, i - h + 2k)$. We show by induction on i that $q_{\mathrm{L}i} = q_{\mathrm{R}i}$ for $0 \leq i \leq h - k$. Since v_k is invertible, (8) and (9) give

$$q_{\mathrm{L}h-k} = q_{\mathrm{R}h-k} = v_k^{-1} \tag{10}$$

and

$$q_{\mathrm{L}i} = -\sum_{j=M}^{k-1} v_k^{-1}\, v_j\, q_{\mathrm{L}i+k-j} \quad q_{\mathrm{R}i} = -\sum_{j=M}^{k-1} q_{\mathrm{R}i+k-j}\, v_j\, v_k^{-1}, \quad 0 \leq i < h-k. \tag{11}$$

Equation (10) gives the base of the induction. Now suppose $q_{\mathrm{L}i} = q_{\mathrm{R}i}$ for $N < i \leq h - k$. Then for $i = N \geq 0$ equation (11) gives

$$\begin{aligned}
q_{\mathrm{L}N} &= -\sum_{j=M}^{k-1} v_k^{-1}\, v_j\, q_{\mathrm{L}N+k-j} = -\sum_{j=M}^{k-1} v_k^{-1}\, v_j\, q_{\mathrm{R}N+k-j} \\
&= -\sum_{j=M}^{k-1} v_k^{-1}\, v_j \left(-\sum_{\ell=M}^{k-1} q_{\mathrm{R}N+k-j+k-\ell} v_\ell\, v_k^{-1} \right) \\
&= -\sum_{\ell=M}^{k-1} \left(-\sum_{j=M}^{k-1} v_k^{-1}\, v_j\, q_{\mathrm{R}N+k-j+k-\ell} \right) v_\ell\, v_k^{-1} = -\sum_{\ell=M}^{k-1} q_{\mathrm{R}N+k-j}\, v_\ell\, v_k^{-1} = q_{\mathrm{R}N}.
\end{aligned}$$

□

Thus we may write $\operatorname{shinv}_h v$ without ambiguity in the non-commutative case, *i.e*

$$\operatorname{shinv}_h v = x^h \operatorname{lquo} v = x^h \operatorname{rquo} v. \tag{12}$$

3.3 Quotients from the Whole Shifted Inverse in $R[x]$

We consider computing the left and right quotients in $R[x]$ from the whole shifted inverse. We have the following theorem.

Theorem 6 (Left and right quotients from the whole shifted inverse in $R[x]$) *Let $u, v \in R[x]$, R a ring, with* $\operatorname{degree} v = k$ *and v_k invertible in R. Then for* $h \geq \operatorname{degree} u$*,*

$$\begin{aligned} u \operatorname{lquo} v &= \operatorname{shift}_{-h}(\operatorname{shinv}_h(v) \times u) \quad \textit{and} \\ u \operatorname{rquo} v &= \operatorname{shift}_{-h}(u \times \operatorname{shinv}_h(v)). \end{aligned}$$

Proof. Consider first the right quotient. It is sufficient to show

$$u = \operatorname{shift}_{-h}(u \times \operatorname{shinv}_h v) \times v + r_{\mathrm{R}}$$

for some r_{R} with $\operatorname{degree} r_{\mathrm{R}} < k$. It is therefore sufficient to show

$$\operatorname{shift}_{-k} u = \operatorname{shift}_{-k}\big(\operatorname{shift}_{-h}(u \times \operatorname{shinv}_h v) \times v\big). \tag{13}$$

We have

$$\begin{aligned} (u \times \operatorname{shinv}_h v) \times v &= u \times ((x^h \operatorname{rquo} v) \times v) \qquad (14) \\ &= u \times (x^h - \rho), \quad \rho = 0 \text{ or } \operatorname{degree} \rho < k \\ &= \operatorname{shift}_h u - u \times \rho. \\ \operatorname{shift}_h u &= (u \times \operatorname{shinv}_h v) \times v + u \times \rho. \qquad (15) \end{aligned}$$

Since $h \geq 0$, Theorem 3 applies and equation (15) gives

$$u = \operatorname{shift}_{-h}\big((u \times \operatorname{shinv}_h v) \times v\big) + \operatorname{shift}_{-h}(u \times \rho)$$

with the degree of $\operatorname{shift}_{-h}(u \times \rho)$ less than k. Therefore

$$\begin{aligned} \operatorname{shift}_{-k} u &= \operatorname{shift}_{-k-h}\big((u \times \operatorname{shinv}_h v) \times v\big) \\ &= \operatorname{shift}_{-k}\big(\operatorname{shift}_{-h}(u \times \operatorname{shinv}_h v) \times v\big), \end{aligned}$$

by Theorem 4, and we have shown equation (13) as required. The proof for lquo replaces equation (14) with

$$v \times (\operatorname{shinv}_h v \times u) = (v \times (x^h \operatorname{lquo} v)) \times u$$

and follows the same lines, *mutatis mutandis*. □

As in the commutative case, it may be more efficient to compute only the top part of the product instead of computing the whole thing then shifting away part. Now that we have shown that shift and shinv are well-defined for non-commutative $R[x]$, we next see that shinv may be computed by our generic algorithm.

4 Generic Algorithm for the Whole Shifted Inverse

Earlier work has shown how to compute shinv efficiently for $\mathbb{Z}$, both for Euclidean domains $F[x]$, and generically [10]. The generic version shown here in Algorithm 3. We justify below that it applies equally well to polynomials with non-commutative coefficients. The algorithm operates on a ring D that is required to have a suitable shift and certain other operations and properties must be defined. For example, on $F[x]$, F a field, these are

$$\begin{aligned}
\text{shift}_n\, u &= \begin{cases} u \cdot x^n & \text{if } n \geq 0 \\ u \text{ quo } x^{-n} & \text{if } n < 0 \end{cases} \\
\text{coeff}(u, i) &= u_i \\
\textsc{Shinv0}(v) &= (1/v_k\, x - 1/v_k \cdot v_{k-1} \cdot 1/v_k,\ 2) \\
\textsc{hasCarries} &= \text{false} \\
\textsc{Mult}(a, b) &= ab \\
\textsc{MultMod}(a, b, n) &= ab \text{ rem } x^n.
\end{aligned}$$

The iterative step of Algorithm 3 is given on line 32. Since D.PowDiff computes $\text{shift}_h\, 1 - v \cdot w$, this line computes

$$\text{shift}_m\, w + \text{shift}_{2m-h}\, \big(w \cdot (\text{shift}_h\, 1 - v \cdot w)\big). \tag{16}$$

The shift operations are multiplications by powers of x, with $\text{shift}_h\, p = px^h$. The the expressions involving k, h, ℓ and m for shift amounts arise from multiplication by various powers of x at different points in order to compute shorter polynomials when possible. Since x commutes with all values, it is possible to accumulate these into single pre- and post- shifts. With this in mind, the $R[x]$ operations $+$ and $\cdot$ ultimately compute the polynomial coefficients using the operations of R and the order of the multiplicands in (16) is exactly that of the Newton-Schulz iteration (3). The form of Shinv0 above is chosen so that it gives a suitable initial value for non-commutative polynomials.

The computational complexity of the Refine methods of Algorithm 3 may be summarized as follows: The function D.Refine1 computes full-length values at each iteration so has time complexity $O(\log(h-k)M(h))$ where $M(N)$ is the time complexity of multiplication. The function D.Refine2 reduces the size of the values, computing only the necessary prefixes. The function D.Refine3 reduces the size of some values further and achieves time complexity $O\big(\sum_{i=1}^{\log(h-k)} M(2^i)\big)$, which gives time complexity $O(M(N)), N = h - k$ for the purely theoretical $M(N) \in O(N \log N)$, for Schönhage-Strassen $M(N) \in O(N \log N \log\log N)$ and for $M(N) \in O(N^p), p > 0$.

Algorithm 3 Generic $\textsc{Shinv}(v, h)$

Input: $v \in D, h \in \mathbb{Z}_{>0}$ where $0 < k = \operatorname{prec} v - 1 < h$
Output: $\operatorname{shinv}_h v \in D$

function D.$\textsc{Shinv}$ (v, h)
  ▷ *Domain-specific initialization*
  $(w, \ell) \leftarrow$ D.$\textsc{Shinv0}(v)$ ▷ *Initialize w to ℓ correct places.*
  return D.$\textsc{Refine}(v, h, k, w, \ell)$ ▷ *One of* D.$\textsc{Refine1}$, D.$\textsc{Refine2}$, D.$\textsc{Refine3}$.

▷ *Below, g is the number of guard places and d is the precision doubling shortfall.*
function D.$\textsc{Refine1}$ (v, h, k, w, ℓ)
  if D.$\textsc{HasCarries}$ **then** $g \leftarrow 1;\ d \leftarrow 1$ **else** $g \leftarrow 0;\ d \leftarrow 0$
  $h \leftarrow h + g$
  $w \leftarrow \text{D.}\operatorname{shift}_{h-k-\ell}(w)$ ▷ *Scale initial value to full length*
  while $h - k + 1 - d > \ell$ **do**
    $w \leftarrow$ D.$\textsc{Step}(h, v, w, 0, \ell)$
    $\ell \leftarrow \min(2\ell - d, h - k + 1 - d)$ ▷ *Number of accurate digits*
  return w

function D.$\textsc{Refine2}$ (v, h, k, w, ℓ)
  if D.$\textsc{HasCarries}$ **then** $g \leftarrow 2;\ d \leftarrow 1$ **else** $g \leftarrow 0;\ d \leftarrow 0$
  $w \leftarrow \text{D.}\operatorname{shift}_g w$
  while $h - k + 1 - d > \ell$ **do**
    $m \leftarrow \min(h - k + 1 - \ell, \ell)$ ▷ *How much to grow*
    $w \leftarrow \text{D.}\operatorname{shift}_{-d}$ D.$\textsc{Step}\big(k + \ell + m + d - 1 + g,\ v,\ w\ , m,\ \ell - g\big)$
    $\ell \leftarrow \ell + m - d$
  return w

function D.$\textsc{Refine3}$ (v, h, k, w, ℓ)
  if D.$\textsc{HasCarries}$ **then** $g \leftarrow 2;\ d \leftarrow 1$ **else** $g \leftarrow 0;\ d \leftarrow 0$
  $w \leftarrow \text{D.}\operatorname{shift}_g w$
  while $h - k + 1 - d > \ell$ **do**
    $m \leftarrow \min(h - k + 1 - \ell,\ \ell)$
    $s \leftarrow \max(0,\ k - 2\ell + 1 - g)$
    $w \leftarrow \text{D.}\operatorname{shift}_{-d}\big(\text{D.}\textsc{Step}\big(k + \ell + m - s - 1 + d + g,\ \text{D.}\operatorname{shift}_{-s} v,\ w,\ m,\ \ell - g\big)\big)$
    $\ell \leftarrow \ell + m - d$
  return $\text{D.}\operatorname{shift}_{-g}(w)$

function D.$\textsc{Step}$ (h, v, w, m, ℓ)
  $\text{D.}\operatorname{shift}_m w \ + \text{D.}\operatorname{shift}_{2m-h}$ $\textsc{Mult}\big(w, \text{D.}\textsc{PowDiff}(v, w, h - m, \ell)\big)$

▷ *Compute* $\text{D.}\operatorname{shift}_h 1 - vw$ *efficiently.*
function D.$\textsc{PowDiff}$ (v, w, h, ℓ)
  $c \leftarrow$ **if** D.$\textsc{HasCarries}$ **then** 1 **else** 0
  $L \leftarrow \text{D.}\operatorname{prec} v + \text{D.}\operatorname{prec} w\ - \ell + c$ ▷ *c for coeff to peek*
  if $v = 0 \vee w = 0 \vee L \geq h$ **then**
    return $\text{D.}\operatorname{shift}_h 1 -$ D.$\textsc{Mult}(v, w)$
  else
    $P \leftarrow$ D.$\textsc{MultMod}(v, w, L)$
    if D.$\textsc{HasCarries}$ $\wedge$ $\text{D.coeff}(P, L - 1) \neq 0$ **then return** $\text{D.}\operatorname{shift}_L 1 - P$
    else return $-P$

5 Non-Commutative Polynomial Example

We give an example of computing left and right quotients via the whole shifted inverse with $R[x] = {F_7}^{2\times 2}[x]$ using the algorithms of Sections 3 and 4. Note that $R[x]$ is not a domain—there may be zero divisors, but it is easy enough to check for them. This example, and the one in Section 7, were produced using the `Domains` package in Maple [5]. The setup to use the `Domains` package for this example is

```
with(Domains);
F     := GaloisField(7);
F2x2  := SquareMatrix(2, F);
PF2x2 := DenseUnivariatePolynomial(F2x2, x);
```

We start with

$$u = \begin{bmatrix} 4 & 6 \\ 6 & 1 \end{bmatrix} x^5 + \begin{bmatrix} 2 & 2 \\ 0 & 1 \end{bmatrix} x^4 + \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix} x^3 + \begin{bmatrix} 2 & 0 \\ 4 & 1 \end{bmatrix} x^2 + \begin{bmatrix} 3 & 3 \\ 5 & 4 \end{bmatrix} x + \begin{bmatrix} 4 & 5 \\ 1 & 2 \end{bmatrix},$$
$$v = \begin{bmatrix} 4 & 3 \\ 4 & 5 \end{bmatrix} x^2 + \begin{bmatrix} 5 & 3 \\ 0 & 4 \end{bmatrix} x + \begin{bmatrix} 1 & 2 \\ 6 & 1 \end{bmatrix}.$$

The whole 5-shifted inverse of v is then

$$\operatorname{shinv}_5 v = \begin{bmatrix} 5 & 4 \\ 3 & 4 \end{bmatrix} x^3 + \begin{bmatrix} 6 & 0 \\ 4 & 1 \end{bmatrix} x^2 + \begin{bmatrix} 1 & 0 \\ 2 & 2 \end{bmatrix} x + \begin{bmatrix} 5 & 1 \\ 6 & 3 \end{bmatrix}.$$

From this, the left and right quotients and remainders are computed to be

$$q_{\text{L}} = \begin{bmatrix} 2 & 6 \\ 1 & 1 \end{bmatrix} x^3 + \begin{bmatrix} 6 & 1 \\ 0 & 0 \end{bmatrix} x^2 + \begin{bmatrix} 2 & 0 \\ 3 & 3 \end{bmatrix} x + \begin{bmatrix} 3 & 1 \\ 0 & 0 \end{bmatrix}, \qquad r_{\text{L}} = \begin{bmatrix} 1 & 6 \\ 4 & 1 \end{bmatrix} x + \begin{bmatrix} 1 & 4 \\ 4 & 3 \end{bmatrix},$$
$$q_{\text{R}} = \begin{bmatrix} 3 & 5 \\ 5 & 0 \end{bmatrix} x^3 + \begin{bmatrix} 1 & 1 \\ 1 & 5 \end{bmatrix} x^2 + \begin{bmatrix} 0 & 5 \\ 5 & 5 \end{bmatrix} x + \begin{bmatrix} 4 & 0 \\ 2 & 6 \end{bmatrix}, \qquad r_{\text{R}} = \begin{bmatrix} 2 & 0 \\ 2 & 1 \end{bmatrix} x + \begin{bmatrix} 0 & 4 \\ 5 & 6 \end{bmatrix}.$$

Taking a larger example where u has degree 100 and v degree 10, D.Refine1 computes $\operatorname{shinv}_{100} v$ with one guard digit in 6 steps with intermediate values of w all of prec 92. Methods D.Refine2 and D.Refine3 compute the same result also in 6 steps but with values of w have prec 4, 8, 16, 32, 64, 92 successively. Method D.Refine3 uses a shorter prefix of v on the first iteration ($s = 3$). The Maple code used for this example is given in Figure 1.

6 Division in $R[x;\sigma,\delta]$

We now examine the more general case where the polynomial variable does not commute with coefficients. For quotients and remainders to be defined, a notion of degree is required and we note that this leads immediately to Ore extensions, or skew polynomials. After touching upon classical algorithms, we introduce the notions of left and right whole shifted inverse. We note that the modified Newton-Schulz iteration may be used to compute whole shifted inverses, though in this case there is no benefit over classical division. Finally, we show how left and right whole shifted inverses may be used to compute right and left quotients, each with only one multiplication.

6.1 Definitions and Classical Algorithms

Consider a ring of objects with elements from a ring R extended by x, with x not necessarily commuting with elements of R. By distributivity, any finite expression in this extended ring is equal to a sum of monomials, the monomials composed of products of elements of R and x. To have a well-defined degree compatible with that of usual polynomials, it is required that

$$\forall\, r \in R \;\exists\, a,b,c,d \in R \;\text{ s.t. }\; xr - rx = ax + b = xc + d. \tag{17}$$

We call the elements of such a ring skew polynomials. Condition (17) implies that for all $r \in R$ there exist $\sigma(r), \delta(r) \in R$ such that

$$x\, r = \sigma(r)\, x + \delta(r). \tag{18}$$

Therefore, to have well-defined notion of degree, the ring must be an Ore extension, $R[x;\delta,\sigma]$. Ore studied these non-commutative polynomials almost a century ago [6] and overviews of Ore extensions in computer algebra are given in [1,2]. The subject is viewed from a linear algebra perspective in [3] and the complexity of skew arithmetic is studied in [9]. The ring axioms of $R[x;\sigma,\delta]$ imply that σ be an endomorphism on R and δ be a σ-derivation, *i.e.* for all $r, s \in R$

$$\delta(r+s) = \delta(r) + \delta(s) \qquad\qquad \delta(r \cdot s) = \sigma(r) \cdot \delta(s) + \delta(r) \cdot s.$$

Different choices of σ and δ allow skew polynomials to represent linear differential operators, linear difference operators, q-generalizations of these and other algebraic systems.

Condition (18) implies that it is possible to write any skew polynomial as a sum of monomials with all the powers of x on the right or all on the left. We will use the notation u_i for coefficients of skew polynomials with all powers of the variable on the right and ${}_iu$ for coefficients with all powers of the variable on the left, *e.g.*

$$u = \sum_{i=0}^{h} u_i x^i = \sum_{i=0}^{h} x^i\, {}_iu.$$

Algorithm 4 Classical division for $R[x;\sigma,\delta]$ with invertible v_k

1: ▷ *Compute q and r from u of degree h and v of degree k such that $u = q \times_\pi v + r$. The left division algorithm applies when σ is bijective.*

2: **function** SKEWDIV $(u, v \in R[x;\sigma,\delta], \pi \in S_2,$ QCOEFF$)$
3: $\quad v^* \leftarrow \text{inv } v_k$
4: $\quad q \leftarrow 0;\ r \leftarrow u$
5: $\quad$ **for** $i \leftarrow h-k$ to 0 by -1 **do**
6: $\quad\quad t \leftarrow$ QCOEFF$(r_{i+k}, v^*, i, k) \times x^i$
7: $\quad\quad q \leftarrow q + t\ ;\ r \leftarrow r - t \times_\pi v$
8: $\quad$ **return** (q, r)

9: ▷ *Left division:* $(q_{\text{L}}, r_{\text{L}}) \leftarrow$ LSKEWDIV$(u, v) \Rightarrow u = v \times q_{\text{L}} + r_{\text{L}}$
10: LSKEWDIV$(u, v) \mapsto$ SKEWDIV$\big(u,\ v,\ (2\,1),\ (a, b, n, k) \mapsto \sigma^{-k}(b \times a)\big)$

11: ▷ *Right division:* $(q_{\text{R}}, r_{\text{R}}) \leftarrow$ RSKEWDIV$(u, v) \Rightarrow u = q_{\text{R}} \times v + r_{\text{R}}$
12: RSKEWDIV$(u, v) \mapsto$ SKEWDIV$\big(u,\ v,\ (1\,2),\ (a, b, n, k) \mapsto a \times \sigma^n(b)\big)$

Algorithm 4 gives left and right classical division in $R[x;\sigma,\delta]$. As in Section 3, $\times_\pi$ is multiplication with arguments permuted by π. When $\sigma(r) = r$, $R[x;\sigma,\delta]$ is a differential ring, usually denoted $R[x,\delta]$, and Algorithm 4 specializes to Algorithm 1. The left division algorithm applies only when σ is bijective. If left division is of primary interest, start from $rx = x\sigma^*(r) + \delta^*(r)$ instead of (18) and work in the adjoint ring $R[x;\sigma^*,\delta^*]$.

Some care is needed in Algorithm 4 to avoid duplicating computation. Notice that for RSKEWDIV the application of QCOEFF on line 6 requires n-fold application of σ to $\text{inv } v_k$ and that the computation of $t \times_\pi v$ on line 7 is $\text{coeff}(t)\, x^{i+k} \times v$. The latter requires commuting $h-k$ powers of x across v over the course of the division. Depending on the cost to compute σ, it may be useful to create an array of the values $\sigma^i(\text{inv } v_k)$ for i from 0 to $h-k$. It is also possible to precompute and store the products $x^i \times v$, with $x^{i+1} \times v$ obtained from $x^i \times v$ by one application of (18). Then the $x^i \times v$ may be used in descending order in the **for** loop without re-computation. Both of these pre-computations are performed in the Maple program for `P[RDiv]` shown in Figure 2.

6.2 Whole Shift and Inverse in $R[x;\sigma,\delta]$

It is possible to define left and right analogs of the whole shift and whole shifted inverse for skew polynomials. In general, the left and right operations give different values.

Definition 3 (Left and right whole n-shift in $R[x;\sigma,\delta]$)
Given $u =\in R[x;\sigma,\delta]$ and $n \in \mathbb{Z}$, the left whole n-shift of u *is*

$$\text{lshift}_{n,x}\, u = \sum_{i+n\geq 0} x^{i+n}\, {}_i u,$$

the right whole n-shift of u *is*

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

When x is clear by context, we write $\operatorname{lshift}_n u$ *and* $\operatorname{rshift}_n u$.

Definition 4 (Left and right whole n-shifted inverse in $R[x;\sigma,\delta]$) *Given $n \in \mathbb{Z}_{\geq 0}$ and $v \in R[x;\sigma,\delta]$, the* left whole n-shifted inverse of v with respect to x *is*

$$\operatorname{lshinv}_{n,x} v = x^n \operatorname{lquo} v$$

the right whole n-shifted inverse of v with respect to x *is*

$$\operatorname{rshinv}_{n,x} v = x^n \operatorname{rquo} v$$

When x is clear by context, we write $\operatorname{lshinv}_n v$ *and* $\operatorname{rshinv}_n v$.

Modified Newton-Schulz Iteration For monic $v \in R[x;\sigma,\delta]$, the whole shifted inverses may be computed using modified Newton-Schulz iterations with $g = 1$ guard places as follows:

$$\begin{aligned}
w_{\mathrm{L}(0)} &= w_{\mathrm{R}(0)} = x^{h-k+g} - v_{k-1}x^{h-k-1+g} \\
w_{\mathrm{L}(i+1)} &= w_{\mathrm{L}(i)} + \operatorname{rshift}_{-h}\left(w_{\mathrm{L}(i)} \times (\operatorname{rshift}_h 1 - v \times w_{\mathrm{L}(i)})\right), \\
w_{\mathrm{R}(i+1)} &= w_{\mathrm{R}(i)} + \operatorname{lshift}_{-h}\left((\operatorname{lshift}_h 1 - w_{\mathrm{R}(i)} \times v) \times w_{\mathrm{R}(i)}\right), \\
&\operatorname{rshift}_{-g} w_{\mathrm{L}(i)} \to \operatorname{lshinv}_h v \\
&\operatorname{lshift}_{-g} w_{\mathrm{R}(i)} \to \operatorname{rshinv}_h v.
\end{aligned} \tag{19}$$

These generalize D.Refine1 in Algorithm 3. For D.Refine2 and D.Refine3, the shifts that reduce the size of intermediate expressions are combined into one pre- and one post-shift in $R[x]$. But on $R[x;\sigma,\delta]$ we do not expect these simplifications of shift expressions to be legitimate.

Even though (19) *can* be used to compute whole shifted inverses, it does not give any benefit over classical division. In the special case of $R[x,\delta]$, the multiplication by v and then by w make it so each iteration creates only one correct term, so $h-k$ iterations are required rather than $\log_2(h-k)$. In other skew polynomial rings, *e.g.* linear difference operators, the iteration (19) can still converge, but with multiple iterations required for each degree of the quotient. It is therefore simpler to compute lshinv and rshinv by classical division.

6.3 Quotients from Whole Shifted Inverses in $R[x;\sigma,\delta]$

It is possible to compute left and right quotients from the right and left whole shifted inverses in $R[x;\sigma,\delta]$. Although computing whole shifted inverses is not asymptotically fast as it is in $R[x]$, once a whole shifted inverse is obtained it can be used to compute multiple quotients and hence remainders, each requiring only one multiplication. This is useful, *e.g.*, when working with differential ideals. In some cases this multiplication of skew polynomials is asymptotically fast [8].

Theorem 7 (Quotients from whole shifted inverses in $R[x;\sigma,\delta]$)
Let $u, v \in R[x;\sigma,\delta]$, with R a ring, $k = \deg v$, $h = \deg u$, and v_k invertible in R. Then

$$u \operatorname{rquo} v = \operatorname{rshift}_{-h}(u \times \operatorname{lshinv}_h v) \tag{20}$$
$$u \operatorname{lquo} v = \operatorname{lshift}_{-h}(\operatorname{rshinv}_h v \times u). \tag{21}$$

Proof. We first prove (20). For $h \geq k$, we proceed by induction on $h-k$. Suppose $h-k=0$. Since $u - (u_h \times 1/v_k) \times v$ has no term of degree h, we have

$$u \operatorname{rquo} v = u_h \times 1/v_k.$$

On the other hand, when $h = k$, $\operatorname{lshinv}_h v = 1/v_k$ so

$$\operatorname{rshift}_{-h}(u \times \operatorname{lshinv}_h v) = u_h \times 1/v_k$$

and (20) holds. For the inductive step, we assume that (20) holds for $h-k<N$. For $h-k=N$, let $u = q \times v + o(x^k)$ and let Q, $\hat{q}$ and $\hat{u}$ be given by

$$u = (Qx^{h-k} + \hat{q}) \times v + r, \qquad Q \in R,\ \ \hat{q} \in o(x^{h-k}),\ \ r \in o(x^k),$$
$$\hat{u} = u - Qx^{h-k} \times v.$$

With this, $\hat{u}$ has degree at most $h-1$. The inductive hypothesis gives $\hat{u} \operatorname{rquo} v = \operatorname{rshift}_{-h}(\hat{u} \times \operatorname{lshinv}_h v)$. Therefore,

$$\begin{aligned}
\hat{u} = u - Qx^{h-k} \times v &= (\hat{u} \operatorname{rquo} v) \times v + \hat{r}, \quad \hat{r} \in o(x^k) \\
&= \operatorname{rshift}_{-h}(\hat{u} \times \operatorname{lshinv}_h v) \times v + \hat{r} \\
\Rightarrow \quad u &= \left(\operatorname{rshift}_{-h}(\hat{u} \times \operatorname{lshinv}_h v) + Qx^{h-k}\right) \times v + \hat{r} \\
&= \operatorname{rshift}_{-h}(\hat{u} \times \operatorname{lshinv}_h v + Qx^{2h-k}) \times v + \hat{r}.
\end{aligned}$$

From this, we have

$$\begin{aligned}
u \operatorname{rquo} v &= \operatorname{rshift}_{-h}(\hat{u} \times \operatorname{lshinv}_h v + Qx^{2h-k}) \\
&= \operatorname{rshift}_{-h}\left((u - Qx^{h-k} \times v) \times \operatorname{lshinv}_h v + Qx^{2h-k}\right) \\
&= \operatorname{rshift}_{-h}\left(u \times \operatorname{lshinv}_h v - Qx^{h-k} \times v \times \operatorname{lshinv}_h v + Qx^{2h-k}\right) \\
&= \operatorname{rshift}_{-h}\left(u \times \operatorname{lshinv}_h v - Qx^{h-k} \times v \times (x^h \operatorname{lquo} v) + Qx^{2h-k}\right) \\
&= \operatorname{rshift}_{-h}\left(u \times \operatorname{lshinv}_h v - Qx^{h-k} \times (x^h + o(x^k)) + Qx^{2h-k}\right) \\
&= \operatorname{rshift}_{-h}\left(u \times \operatorname{lshinv}_h v + Q \times o(x^h)\right) = \operatorname{rshift}_{-h}(u \times \operatorname{lshinv}_h v).
\end{aligned}$$

This completes the inductive step and the proof of (20). Equation (21) is proven as above, *mutatis mutandis*. □

As in the commutative case, it may be more efficient to compute only the required top part of the product in (20) and (21) rather than to compute the whole product and then shift by $-h$.

7 Skew Polynomial Examples

7.1 Differential Operators

We take $F_7[y, \partial_y]$ as a first example of using whole shifted inverses to compute quotients of skew polynomials. We use Algorithm 4 to compute the left and right whole shifted inverses, and then Theorem 7 to obtain the quotients. We start with u and v

$$\begin{aligned} u &= (3y+6)\partial_y^5 + (3y+1)\partial_y^4 + 6y\partial_y^3 + 4y\partial_y^2 + (2y+1)\partial_y + (2y+5) \\ v &= 4\partial_y^2 + (2y+5)\partial_y + (4y+6). \end{aligned}$$

The whole shifted inverses $\mathrm{lshinv}_5\, v = \partial_y^5 \,\mathrm{lquo}\, v$ and $\mathrm{rshinv}_5 = \partial_y^5 \,\mathrm{rquo}\, v$ are computed by Algorithm 4.

$$\begin{aligned} \mathrm{lshinv}_5 &= 2\partial_y^3 + (6y+1)\partial_y^2 + (4y^2+4y+3)\partial_y + (5y^3+y^2+3y+2) \\ \mathrm{rshinv}_5 &= 2\partial_y^3 + (6y+1)\partial_y^2 + (4y^2+4y+5)\partial_y + (5y^3+y^2+y+1). \end{aligned}$$

Then $q_{\mathrm{L}} = \mathrm{lshift}_{-5}(\mathrm{rshinv}_5\, v \times u)$ and $q_{\mathrm{R}} = \mathrm{rshift}_{-5}(u \times \mathrm{lshinv}_5\, v)$ so

$$\begin{aligned} q_{\mathrm{L}} &= (6y+5)\partial_y^3 + (4y^2+3y+3)\partial_y^2 + (5y^3+5y^2+5)\partial_y \\ &\quad + (y^4+3y^3+5y^2+5y+2) \\ r_{\mathrm{L}} &= (5y^5+4y^4+3y^3+6y^2+4y)\partial_y + (3y^5+2y^4+y^3+5y^2+5) \end{aligned}$$

$$\begin{aligned} q_{\mathrm{R}} &= (6y+5)\partial_y^3 + (4y^2+3y+1)\partial_y^2 + (5y^3+5y^2+4y+3)\partial_y \\ &\quad + (y^4+3y^3+5y^2+3y+5) \\ r_{\mathrm{R}} &= (5y^5+4y^4+6y^3)\partial_y + (3y^5+3y^4+5y^3+y^2+4y+5). \end{aligned}$$

A proof-of-concept Maple implementation for generic skew polynomials is given in Figure 2. The program is to clarify any ambiguities without any serious attention to efficiency. The setup for the above example is

```
with(Domains):
LinearOrdinaryDifferentialOperator :=
    (R, x) -> SkewPolynomial(R, x, r->r, R[Diff], r->r):

F    := GaloisField(7):
R    := DenseUnivariatePolynomial(F, 'y'):
Lodo := LinearOrdinaryDifferentialOperator(R, 'D[y]'):
```

7.2 Difference Operators

We use linear ordinary difference operators as a second example, this time with σ not being the identity. We construct $F_7[y, \Delta_y]$ as $F_7[y][\Delta_y; E, E-1]$. As before, we use Algorithm 4 to compute the left and right whole shifted inverses, and then Theorem 7 to obtain the quotients. We take u and v to be

$$u = y\Delta_y^5 + (3y+6)\Delta_y^4 + (6y+5)\Delta_y^3 + 3y\Delta_y^2 + (2y+1)\Delta_y + 5y$$
$$v = 4\Delta_y^2 + (6y+1)\Delta_y + (6y+6).$$

The whole shifted inverses $\mathrm{lshinv}_5\, v = \Delta_y^5 \,\mathrm{lquo}\, v$ and $\mathrm{rshinv}_5 = \Delta_y^5 \,\mathrm{rquo}\, v$ are computed by Algorithm 4.

$$\mathrm{lshinv}_5 = 2\Delta_y^3 + (4y+2)\Delta_y^2 + (y^2+4y)\Delta_y + (2y^3+6y^2+y)$$
$$\mathrm{rshinv}_5 = 2\Delta_y^3 + (4y+1)\Delta_y^2 + (y^2+2)\Delta_y + (2y^3+y^2+4y+1).$$

Then $q_{\mathrm{L}} = \mathrm{lshift}_{-5}(\mathrm{rshinv}_5\, v \times u)$ and $q_{\mathrm{R}} = \mathrm{rshift}_{-5}(u \times \mathrm{lshinv}_5\, v)$ so

$$\begin{aligned}
q_{\mathrm{L}} &= (2y+3)\Delta_y^3 + (4y^2+3y+4)\Delta_y^2 + (y^3+5y^2+6y+4)\Delta_y \\
&\quad + (2y^4+6y^3+4y^2+4y+4) \\
r_{\mathrm{L}} &= (2y^5+6y^4+6y^2+5y+3)\Delta_y + (2y^5+2y^4+4y^3+2y+1) \\
q_{\mathrm{R}} &= 2y\Delta_y^3 + (4y^2+5)\Delta_y^2 + (y^3+5y^2+y+6)\Delta_y + (2y^4+4y^3+5y+1) \\
r_{\mathrm{R}} &= (2y^5+3y^4+4y^3+y^2)\Delta_y + (2y^5+6y^4+5y^3+3y^2+5y).
\end{aligned}$$

The Maple setup for this example is

```
# Delta(f) acts as  subs(y=y+1, f) - f  for f in R
LinearOrdinaryDifferenceOperator := proc(R, x, C)
    local E := R[ShiftOperator];
    SkewPolynomial(R, x, r->E(r,C[1]), r->R[`-`](E(r,C[1]),r),
                         r->E(r,C[`-`](C[1])));
end:

F    := GaloisField(7);
R    := DenseUnivariatePolynomial(F, 'y');
Lodo := LinearOrdinaryDifferenceOperator(R, 'Delta[y]', F)
```

7.3 Difference Operators with Matrix Coefficients

As a final example, we take quotients in $F_7^{2\times 2}[y, \Delta_y]$ to underscore the genericity of this method.

$$\begin{aligned}
u = &\left(\begin{bmatrix}6&0\\1&1\end{bmatrix} y + \begin{bmatrix}3&0\\2&0\end{bmatrix}\right)\Delta_y^5 + \left(\begin{bmatrix}4&4\\6&5\end{bmatrix} y + \begin{bmatrix}3&2\\4&4\end{bmatrix}\right)\Delta_y^4 + \left(\begin{bmatrix}4&3\\0&3\end{bmatrix} y + \begin{bmatrix}1&1\\4&1\end{bmatrix}\right)\Delta_y^3 \\
&+ \left(\begin{bmatrix}0&1\\4&5\end{bmatrix} y + \begin{bmatrix}3&2\\5&4\end{bmatrix}\right)\Delta_y^2 + \left(\begin{bmatrix}0&6\\4&3\end{bmatrix} y + \begin{bmatrix}0&0\\0&6\end{bmatrix}\right)\Delta_y + \left(\begin{bmatrix}5&3\\6&2\end{bmatrix} y + \begin{bmatrix}5&2\\1&2\end{bmatrix}\right)
\end{aligned}$$

$$v = \begin{bmatrix}1&5\\2&6\end{bmatrix}\Delta_y^2 + \left(\begin{bmatrix}1&5\\0&0\end{bmatrix} y + \begin{bmatrix}4&6\\3&4\end{bmatrix}\right)\Delta_y + \left(\begin{bmatrix}2&6\\0&4\end{bmatrix} y + \begin{bmatrix}0&3\\1&2\end{bmatrix}\right)$$

$$\mathrm{lshinv}_5 = \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix} \Delta_y^3 + \left(\begin{bmatrix} 5 & 0 \\ 3 & 0 \end{bmatrix} y + \begin{bmatrix} 0 & 4 \\ 1 & 2 \end{bmatrix} \right) \Delta_y^2 + \left(\begin{bmatrix} 2 & 0 \\ 4 & 0 \end{bmatrix} y^2 + \begin{bmatrix} 3 & 1 \\ 0 & 1 \end{bmatrix} y + \begin{bmatrix} 0 & 2 \\ 4 & 4 \end{bmatrix} \right) \Delta_y$$
$$+ \left(\begin{bmatrix} 5 & 0 \\ 3 & 0 \end{bmatrix} y^3 + \begin{bmatrix} 4 & 2 \\ 0 & 4 \end{bmatrix} y^2 + \begin{bmatrix} 2 & 6 \\ 6 & 6 \end{bmatrix} y + \begin{bmatrix} 1 & 2 \\ 6 & 6 \end{bmatrix} \right)$$

$$\mathrm{rshinv}_5 = \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix} \Delta_y^3 + \left(\begin{bmatrix} 5 & 0 \\ 3 & 0 \end{bmatrix} y + \begin{bmatrix} 4 & 4 \\ 2 & 2 \end{bmatrix} \right) \Delta_y^2 + \left(\begin{bmatrix} 2 & 0 \\ 4 & 0 \end{bmatrix} y^2 + \begin{bmatrix} 2 & 1 \\ 5 & 1 \end{bmatrix} y + \begin{bmatrix} 6 & 0 \\ 0 & 2 \end{bmatrix} \right) \Delta_y$$
$$+ \left(\begin{bmatrix} 5 & 0 \\ 3 & 0 \end{bmatrix} y^3 + \begin{bmatrix} 2 & 2 \\ 3 & 4 \end{bmatrix} y^2 + \begin{bmatrix} 3 & 5 \\ 5 & 4 \end{bmatrix} y + \begin{bmatrix} 1 & 3 \\ 3 & 1 \end{bmatrix} \right)$$

$$q_{\mathrm{L}} = \left(\begin{bmatrix} 1 & 3 \\ 1 & 5 \end{bmatrix} y + \begin{bmatrix} 3 & 1 \\ 6 & 4 \end{bmatrix} \right) \Delta_y^3 + \left(\begin{bmatrix} 2 & 0 \\ 4 & 0 \end{bmatrix} y^2 + + \begin{bmatrix} 4 & 6 \\ 2 & 1 \end{bmatrix} y + \begin{bmatrix} 2 & 1 \\ 5 & 0 \end{bmatrix} \right) \Delta_y^2$$
$$+ \left(\begin{bmatrix} 5 & 0 \\ 3 & 0 \end{bmatrix} y^3 + \begin{bmatrix} 4 & 0 \\ 6 & 6 \end{bmatrix} y^2 + \begin{bmatrix} 2 & 4 \\ 5 & 4 \end{bmatrix} y + \begin{bmatrix} 0 & 5 \\ 6 & 1 \end{bmatrix} \right) \Delta_y$$
$$+ \left(\begin{bmatrix} 2 & 0 \\ 4 & 0 \end{bmatrix} y^4 + \begin{bmatrix} 4 & 3 \\ 2 & 6 \end{bmatrix} y^3 + \begin{bmatrix} 1 & 0 \\ 5 & 0 \end{bmatrix} y^2 + \begin{bmatrix} 4 & 3 \\ 1 & 5 \end{bmatrix} y + \begin{bmatrix} 5 & 6 \\ 1 & 6 \end{bmatrix} \right)$$

$$r_{\mathrm{L}} = \left(\begin{bmatrix} 6 & 0 \\ 0 & 0 \end{bmatrix} y^5 + \begin{bmatrix} 6 & 2 \\ 1 & 0 \end{bmatrix} y^4 + \begin{bmatrix} 6 & 6 \\ 4 & 6 \end{bmatrix} y^3 + \begin{bmatrix} 2 & 2 \\ 3 & 6 \end{bmatrix} y^2 + \begin{bmatrix} 2 & 4 \\ 6 & 0 \end{bmatrix} y + \begin{bmatrix} 6 & 5 \\ 2 & 0 \end{bmatrix} \right) \Delta_y$$
$$+ \left(\begin{bmatrix} 0 & 0 \\ 5 & 0 \end{bmatrix} y^5 + \begin{bmatrix} 6 & 0 \\ 3 & 4 \end{bmatrix} y^4 + \begin{bmatrix} 3 & 2 \\ 3 & 6 \end{bmatrix} y^3 + \begin{bmatrix} 5 & 1 \\ 3 & 0 \end{bmatrix} y^2 + \begin{bmatrix} 3 & 6 \\ 4 & 6 \end{bmatrix} y + \begin{bmatrix} 2 & 4 \\ 2 & 6 \end{bmatrix} \right)$$

$$q_{\mathrm{R}} = \left(\begin{bmatrix} 5 & 4 \\ 6 & 1 \end{bmatrix} y + \begin{bmatrix} 6 & 2 \\ 4 & 6 \end{bmatrix} \right) \Delta_y^3 + \left(\begin{bmatrix} 2 & 0 \\ 1 & 0 \end{bmatrix} y^2 + \begin{bmatrix} 0 & 0 \\ 6 & 0 \end{bmatrix} y + \begin{bmatrix} 5 & 3 \\ 4 & 5 \end{bmatrix} \right) \Delta_y^2$$
$$+ \left(\begin{bmatrix} 5 & 0 \\ 6 & 0 \end{bmatrix} y^3 + \begin{bmatrix} 1 & 6 \\ 0 & 2 \end{bmatrix} y^2 + \begin{bmatrix} 5 & 5 \\ 1 & 4 \end{bmatrix} y + \begin{bmatrix} 5 & 3 \\ 2 & 6 \end{bmatrix} \right) \Delta_y$$
$$+ \left(\begin{bmatrix} 2 & 0 \\ 1 & 0 \end{bmatrix} y^4 + \begin{bmatrix} 2 & 5 \\ 5 & 6 \end{bmatrix} y^3 + \begin{bmatrix} 5 & 2 \\ 4 & 3 \end{bmatrix} y^2 + \begin{bmatrix} 2 & 2 \\ 1 & 1 \end{bmatrix} y + \begin{bmatrix} 2 & 5 \\ 2 & 3 \end{bmatrix} \right)$$

$$r_{\mathrm{R}} = \left(\begin{bmatrix} 5 & 4 \\ 6 & 2 \end{bmatrix} y^5 + \begin{bmatrix} 1 & 4 \\ 0 & 3 \end{bmatrix} y^4 + \begin{bmatrix} 4 & 4 \\ 3 & 2 \end{bmatrix} y^3 + \begin{bmatrix} 1 & 3 \\ 1 & 4 \end{bmatrix} y^2 + \begin{bmatrix} 3 & 2 \\ 2 & 5 \end{bmatrix} y + \begin{bmatrix} 2 & 6 \\ 4 & 5 \end{bmatrix} \right) \Delta_y$$
$$+ \left(\begin{bmatrix} 3 & 2 \\ 5 & 1 \end{bmatrix} y^5 + \begin{bmatrix} 3 & 4 \\ 4 & 6 \end{bmatrix} y^4 + \begin{bmatrix} 3 & 0 \\ 2 & 6 \end{bmatrix} y^3 + \begin{bmatrix} 6 & 1 \\ 2 & 6 \end{bmatrix} y^2 + \begin{bmatrix} 3 & 2 \\ 6 & 0 \end{bmatrix} y + \begin{bmatrix} 4 & 0 \\ 1 & 3 \end{bmatrix} \right)$$

The Maple setup for this example is the same as for the previous example but with `F := SquareMatrix(2, GaloisField(7))`.

8 Conclusions

We have extended earlier work on efficient computation of quotients in a generic setting to the case of non-commutative univariate polynomial rings. We have shown that when the polynomial variable commutes with the coefficients, the whole shift and whole shifted inverse are well-defined and they may be used to compute left and right quotients. The whole shifted inverse may be computed by a modified Newton method in exactly the same way as when the coefficients are commutative and the number of iterations is logarithmic in the degree of the result. When the polynomial variable does not commute with the coefficients, left and right whole shifted inverses exist and may be computed by classical division. Once a left or right whole shifted inverse is obtained, several right or left quotients with that divisor may be computed, each with a single multiplication.

References

1. Abramov, S.A., Le, H.Q., Li, Z.: Univariate Ore polynomial rings in computer algebra. Journal of Mathematical Sciences **131**(5), 5885–5903 (2005)
2. Bronstein, M., Petkovšek, M.: An introduction to pseudo-linear algebra. Theoretical Computer Science **157**(1), 3–33 (1996)
3. Jacobson, N.: Pseudo-linear transformations. Annals of Mathematics, Second Series **38**(2), 484–507 (1937)
4. Moenck, R.T., Borodin, A.B.: Fast modular transforms via division. In: Proc. 13th Annual Symposium on Switching and Automata Theory (SWAT 1972). pp. 90–96. IEEE, New York (1972)
5. Monagan, M.B.: Gauss: a parameterized domain of computation system with support for signature functions. In: Miola, A. (ed.) Design and Implementation of Symbolic Computation Systems. pp. 81–94. Springer Berlin Heidelberg, Berlin, Heidelberg (1993)
6. Ore, Ø.: Theory of non-commutative polynomials. Annals of Mathematics, Second Series **34**(3), 480–508 (1933)
7. Schulz, G.: Iterative Berechnung der reziproken Matrix. Zeitschrift für Angewandte Mathematik und Mechanik **13**(1), 57–59 (1933)
8. van der Hoeven, J.: FFT-like multiplication of linear differential operators. Journal of Symbolic Computation **33**(1), 123–127 (2002)
9. van der Hoeven, J.: On the complexity of skew arithmetic. Applicable Algebra in Engineering, Communication and Computing **27**, 105–122 (2016)
10. Watt, S.M.: Efficient generic quotients using exact arithmetic. In: Proc. International Symposium on Symbolic and Algebraic Computation (ISSAC 2023). ACM, New York (2023)

```
fshinv := proc (PR, method, h, v, perm)
    local R, x, k, vk, ivk, vkm1, w, ell, m, s, g, rmul, pmul, pshift, monom,
          step, refine, refine1, refine2, refine3;

    R      := PR[CoefficientRing];
    pmul   := (a, b)    -> PR[‘*‘](perm(a, b));
    rmul   := (a, b)    -> R [‘*‘](perm(a, b));
    monom  := (c, x, n) -> PR[‘*‘](PR[Polynom]([c]), PR[‘^‘](x, n));
    pshift := (n,v)     -> shift(PR, n, v);

    step   := proc(h, v, w, m, ell)
        PR[‘+‘]( pshift(m,w), pshift(2*m-h,pmul(w,PR[‘-‘]( PR[‘^‘](x,h-m), pmul(v,w) ))) )
    end;

    refine1 := proc (v, h, k, w0, ell0) local m, s, w, ell;
        w := pshift(h-k-ell0+1, w0); ell := ell0;
        while ell < h-k+1 do
            w := step(h, v, w, 0, ell); ell := min(2*ell, h-k+1)
        od;
        w
    end;
    refine2 := proc (v, h, k, w0, ell0) local m, w, ell;
        w := w0; ell := ell0;
        while ell < h-k+1 do
            m := min(h-k+1-ell, ell);
            w := step(k+ell+m-1, v, w, m, ell); ell := ell+m
        od;
        w
    end;
    refine3 := proc (v, h, k, w0, ell0) local m, s, w, ell;
        w := w0; ell := ell0;
        while ell < h-k+1 do
            m := min(h-k+1-ell, ell); s := max(0, k-2*ell+1);
            w := step(k+ell+m-1-s, pshift(-s, v), w, m, ell); ell := ell+m
        od;
        w
    end;

    if   method = 1 then refine := refine1
    elif method = 2 then refine := refine2
    elif method = 3 then refine := refine3
    else error "Unknown method", method
    fi;

    x   := PR[Polynom]([R[0],R[1]]); k   := PR[Degree](v);
    vk  := PR[Lcoeff](v);            ivk := R[‘^‘](vk, -1);
    if   h < k then return 0
    elif k = 0 or h = k or v = monom(vk,x,k) then return monom(ivk,x,h-k)
    fi;
    vkm1 := PR[Coeff](v, k-1);
    w    := PR[Polynom]([rmul(ivk, rmul(R[‘-‘](vkm1), ivk)), ivk]);  ell := 2;
    g    := 1; # Assume all coeff rings need a guard digit
    pshift(-g, refine(v, h + g, k, w, ell))
end:

fdiv := proc (PR, method, u, v, perm) local mul, h, iv, q, r;
    mul := (a, b) -> PR[‘*‘](perm(a, b));
    h   := PR[Degree](u);
    iv  := fshinv(PR, method, h, v, perm);
    q   := shift(PR,-(h-k),mul(shift(PR,-k,u),iv)); # Need only top h-k terms
    r   := PR[‘-‘](u, mul(q, v));
    (q, r)
end:
lfdiv := (PR, method, u, v) -> fdiv(PR, method, u, v, (a,b)->(b,a)):
rfdiv := (PR, method, u, v) -> fdiv(PR, method, u, v, (a,b)->(a,b)):
```

Fig. 1: Maple code for fast generic polynomial shinv and left and right division

```
SkewPolynomial := proc (R, x, sigma, delta, sigmaInv)
    local P, deltaStar, mult2, MultVarOnLeft, MultVarOnRight;

    # Table to contain the operations.
    P := DenseUnivariatePolynomial(R, x);

    # If x*r = sigma(r)*x + delta(r), then
    #    r*x = x*sigmaInv(r) - delta(sigmaInv(r)) = x*sigmaInv(r) + deltaStar(r)
    deltaStar := r -> R[`-`](delta(sigmaInv(r)));

    P[DomainName]:= 'SkewPolynomial';
    P[Categories]:= P[Categories] minus {CommutativeRing,IntegralDomain};
    P[Properties]:= P[Properties] minus {Commutative(`*`)};

    P[ThetaOp]  := P[Polynom]([R[0], R[1]]);   # The variable as skew polynomial.

    P[Apply] := proc(ell, p) local i, pi, result;   # Apply a skew polynomial as an operator.
        pi     := p;   # delta^i (p)
        result := R[`*`](P[Coeff](ell, 0), pi);
        for i to P[Degree](ell) do  # For Maple, for loop default from is 1.
            pi     := delta(pi);
            result := R[`+`](result, R[`*`](P[Coeff](ell, i), pi))
        od;
        result
    end:

    P[`^`] := proc(a0, n0) local a, n, p;   # Binary powering
        a := a0; n := n0; p := P[1];
        while n > 0 do
          if irem(n,2) = 1 then p := P[`*`](p, a) fi; a := P[`*`](a, a); n := iquo(n,2);
        od;
        p
    end:

    P[`*`] := proc() local i, p;   # N-ary product
        p := P[1]; for i to nargs do p := mult2(p, args[i]) od; p
    end:
    mult2 := proc(a, b) local s, i, ai, xib;   # Binary product
        xib := b;  ai := P[Coeff](a,0);
        s   := P[Map](c->R[`*`](ai, c), xib);
        for i to P[Degree](a) do
            xib := MultVarOnLeft(xib); ai := P[Coeff](a, i);
            s   := P[`+`](s, P[Map](c->R[`*`](ai,c), xib));
        od;
        s
    end:

    # Compute x*b as polynomial with powers on right.
    # x*sum(b[i]*x^i, i=0..degb) = sum(sigma(b[i])*x^(i+1) + delta(b[i])*x^i, i=0..degb)
    MultVarOnLeft := proc(b) local cl, slist, dlist;
        cl    := P[ListCoeffs](b);
        slist := [ R[0], op(map(sigma, cl)) ]; dlist := [ op(map(delta, cl)), R[0] ];
        P[Polynom](zip(R[`+`], slist, dlist));
    end:
    # Compute b*x as polynomial with powers on left.
    # sum(x^i*b[i], i=0..degb)*x = sum(x^(i+1)*sigmaInv(b[i]) + deltaStar(b[i])*x^i, i=0..degb)
    MultVarOnRight := proc(b) local cl, slist, dlist;
        cl    := P[ListCoeffs](b);
        slist := [ R[0], op(map(sigmaInv, cl)) ]; dlist := [ op(map(deltaStar, cl)), R[0] ];
        P[Polynom](zip(R[`+`], slist, dlist));
    end:

    # Continued in Part 2...
```

Fig. 2: Maple code for generic skew polynomials (Part 1)

```
# ... continued from Part 1.

# For v = sum(vr_i x^i, i = 0..k) = sum(x^i vl_i, i = 0..k)
# return polynomial with vl_i, interpreting powers as on left,
# abusing the representation of output.
P[ConvertToAdjointForm] := proc(v) local v_adj, i, rci, rcip;
    v_adj := P[0];
    for i from P[Degree](v) to 0 by -1 do
        rci   := P[Polynom]([P[Coeff](v,i)]);
        v_adj := P[`+`](v_adj, (MultVarOnRight@@i)(rci));
    od;
    v_adj
end:

# For v = sum(x^i vl_i, i = 0..k) = sum(x^i vr_i, i = 0..k)
# return polynomial with vr_i, interpreting powers as on right,
# abusing the representation of input.
P[ConvertFromAdjointForm] := proc(v_adj) local v, i, rci;
    v := P[0];
    for i from 0 to P[Degree](v_adj) do
        rci := P[Polynom]([P[Coeff](v_adj,i)]);
        v   := P[`+`](v, (MultVarOnLeft@@i)(rci))
    od;
    v
end:

# Shift by power on left.
P[LShift] := proc(n, v0) local v, shv, i, k;
    v := P[ConvertToAdjointForm](v0); k := P[Degree](v);
    if k + n < 0 then shv := P[0]
    elif n < 0 then shv := P[Polynom]([seq(P[Coeff](v,i), i = -n..k)])
    else shv := P[Polynom]([seq(R[0], i=1..n), seq(P[Coeff](v,i), i=0..k)])
    fi;
    P[ConvertFromAdjointForm](shv)
end:

# Shift by power on right.
P[RShift] := proc(n, v) local i, k;
    k  := P[Degree](v);
    if k + n < 0 then P[0]
    elif n < 0 then P[Polynom]([seq(P[Coeff](v,i), i = -n..k)])
    else P[Polynom]([seq(R[0], i=1..n), seq(P[Coeff](v,i), i=0..k)])
    fi
end:

# Quotient and remainder
P[GDiv] := proc(perm, qfun) proc (u, v) local h, k, x, ivk, t, q, r, i, qi;
    x   := P[Polynom]([R[0], R[1]]); ivk := R[Inv](P[Lcoeff](v));
    h   := P[Degree](u); k := P[Degree](v);
    q   := P[0];         r := u;
    for i from h - k by -1 to 0 do
        qi := qfun(P[Coeff](r,i+k), ivk, i, k);
        t  := P[`*`](P[Constant](qi), P[`^`](x,i));
        q  := P[`+`](q, t);
        r  := P[`-`](r, P[`*`](perm(t, v)));
    od;
    (q, r)
end end:
P[RDiv0] := P[GDiv](rperm, (u,iv,n,k)->R[`*`](u,(sigma@@n)(iv)));
P[LDiv]  := P[GDiv](lperm, (u,iv,n,k)->(sigmaInv@@k)(R[`*`](iv,u)));

# Continued in Part 3...
```

Fig. 2: Maple code for generic skew polynomials (Part 2)

```
    # ... continued from Part 2.

    # A slightly less repetitive RDiv.
    P[RDiv] := proc (u, v) local h, k, x, ivk, sigma_ivk_i, x_i_v, q, r, i, qi;
        x   := P[Polynom]([R[0], R[1]]); ivk := R[Inv](P[Lcoeff](v));
        h   := P[Degree](u); k := P[Degree](v);

        # Precompute sigma^i(ivk) and x^i*v for required i.
        sigma_ivk_i[0] := ivk;
        for i from 1 to h-k do sigma_ivk_i[i] := sigma(sigma_ivk_i[i-1]); od;
        x_i_v[0] := v;
        for i from 1 to h-k do x_i_v[i]       := P[`*`](x, x_i_v[i-1]) od;

        q   := P[0];   r := u;
        for i from h - k by -1 to 0 do
            qi := P[Constant](R[`*`](P[Coeff](r, i+k), sigma_ivk_i[i]));
            q  := P[`+`](q, P[`*`](qi, P[`^`](x,i)));
            r  := P[`-`](r, P[`*`](qi, x_i_v[i]));
        od;
        (q, r)
    end:

    # Needed for some versions of Maple.
    P[0] := P[Polynom]([R[0]]);
    P[1] := P[Polynom]([R[1]]);
    P[`-`] := proc()
        local nb := P[Polynom](map(c-> R[`-`](c), P[ListCoeffs](args[nargs])));
        if nargs = 1 then nb else P[`+`](args[1], nb) fi
    end:

    # Return the table
    P
end:
```

Fig. 2: Maple code for generic skew polynomials (Part 3)