Two Methods for Efficient Generic Inversion

Stephen M. Watt
Cheriton School of Computer Science, University of Waterloo, Canada
e-mail: smwatt@uwaterloo.ca

Abstract

Two generic methods to compute multiplicative inverses are presented. These methods apply to integers, polynomials and matrices and are asymptotically faster than classical algorithms. The first method is to use a modified Newton iteration to compute quotients via shifted inverses in a not-necessarily commutative Euclidean domain. The second method is to use the Moore-Penrose inverse to avoid pivots and whole-row operations in block matrix inversion.

1. Introduction

In computer algebra it is desirable to find algorithms that can be expressed over abstract algebraic domains and to implement these generically. For example, programs to multiply polynomials in $R[x]$ or perform Gaussian elimination on matrices in $F^{n\times n}$ can be expressed as programs that use the ring and field operations from R and F. This avoids multiple implementations of the same algorithms and allows flexible composition of domains. Systems such as Axiom are built around this principle, and others can support it through mechanisms such as the Maple Domains package. Most modern programming languages, such as C++, C#, Go, Java, Rust and Typescript, support generic programming in one form or another.

In mathematical computing it is required to find quotients or inverses of various types of objects, including integers, polynomials and matrices. In this paper we summarize some earlier results showing how to compute these quotients and inverses efficiently and generically. Section 2 shows a generic algorithm to compute integer and polynomial quotients. This algorithm is both useful in practice and can be asymptotically fast (depending on the multiplication method) and performs all operations without leaving the original domain. Section 3 shows how matrix inverses may be computed on a block representation without breaking the block abstraction.

2. Modified Newton Iteration for Euclidean Domains

On a Euclidean domain D with valuation $N : D \to \mathbb{R}_{\geq 0}$, we define the quotient and remainder of u by v as the unique values q and r such that $u = q \times v + r, \quad N(r) < N(v)$ and write $q = u \operatorname{quo} v$ and $r = u \operatorname{rem} v$. For both integers and polynomials it is well known how to compute quotients efficiently using a Newton iteration. For $u, v \in \mathbb{Z}$, the quotient of u by v may be found by first computing v^{-1} in $\mathbb{R}$ to sufficient precision with a Newton iteration solving $f(x) = 1/x - v = 0$. For $u, v \in F[x]$, F a field, the quotient may be computed in $F[x]/\langle x^{m+1}\rangle$ using Newton iteration to find the inverse of the reverse polynomial $\operatorname{rev}_k v = x^k v(1/x)$, where k and $k+m$ are the degrees of v and u respectively. In both cases, the computation leaves the original domain, which can complicate library structure. In earlier work [8], we have shown how to compute these quotients using only ring operations and shifts with values remaining in the original domain. We summarize those results here.

We define the operations "prec", "shift" and "shinv" on base-B integers and polynomials in x as follows:

Number of coefficients:	$\mathrm{prec}_B(w) = \lfloor \log_B \lvert w \rvert \rfloor + 1$	$\mathrm{prec}_x(p) = \mathrm{degree}_x p + 1$
Whole shift:	$\mathrm{shift}_{n,B}(w) = \lfloor wB^n \rfloor$	$\mathrm{shift}_{n,x}(p) = \sum_{i+n\geq 0} p_i x^{i+n}$
Whole shifted inverse:	$\mathrm{shinv}_n(w) = \lfloor B^n/w \rfloor$	$\mathrm{shinv}_{n,x}(p) = x^n \operatorname{quo} p$

where $p = \sum_{i=0}^{h} p_i x^i$, $n \in \mathbb{Z}$ and $B \in \mathbb{Z}_{\geq 2}$. When the base B or variable x are clear from context, they may be omitted and we simply write shift_n and shinv_n. Depending on the implementation, the shift operation can be performed in time $O(1)$ or $O(n)$. With these definitions, efficient quotients may be computed generically by the following theorems.

Theorem 1. *Let D be $\mathbb{Z}$ or $F[x]$, F a field. Given $u, v \in D$, and $\mathrm{prec}\, u \leq h + 1$,*

$$u \operatorname{quo} v = \mathrm{shift}_{-h}(u \cdot \mathrm{shinv}_h\, v) + \delta, \tag{1}$$

where $\delta = 0$ when $D = F[x]$ and $\delta \in \{0, 1\}$ when $D = \mathbb{Z}$.

Theorem 2. *Let D be $\mathbb{Z}$ or $F[x]$, F a field. Given $v \in D$, $\mathrm{prec}\, v = k + 1 < h + 1$ and suitable starting value $w_{(0)}$, the sequence of iterates*

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

converges to $\mathrm{shinv}_h\, v + \delta$ in $\lceil \log_2(h - k) \rceil$ steps.

The $\delta = 1$ integer case causes no problems, as it is easy to first check whether $u < v + v$. After dispensing with special cases, the shifted inverse of the first two places for integer v may be computed as

$$V := v_k B^2 + v_{k-1} B + v_{k-2} \qquad\qquad w_0 := (B^4 - V) \operatorname{quo} V + 1,$$

assuming $B \geq 16$ and $\mathrm{prec}\ v - 1 = k \geq 2$. The quotient to produce w_0 is obtained by dividing a 4-place quantity by a 2-place quantity. If the base B is less than 16, then digits may be grouped to give a sufficiently large base. For polynomials, the shifted inverse of the first two places of v will be

$$w_0 := x/v_k - v_{k-1}/v_k^2.$$

In both cases, the h-shifted inverse of v may then be computed generically as $\textsc{Shinv}(v, h, k, w_0, 2)$, as shown in Algorithm 1. Note that for polynomials a 1-place initial value would be sufficient to start the iteration, but the generic algorithm takes a simpler form when two places are given. The function HasCarries indicates whether addition can cause carries from one coefficient place to another in the arithmetic of the domain. It gives "true" for integers and "false" for polynomials. The details of this algorithm are justified by the following theorems in the integer case. They relate to iterates of the function $S_{\mathbb{Z}}$, defined as

$$S_{\mathbb{Z}}(h, v, w) := w + \left\lfloor w(B^h - vw)B^{-h} \right\rfloor, \tag{2}$$

which has fixed points at $0, 1, \lfloor B^h/v \rfloor - 1$ and $\lfloor B^h/v \rfloor$. Together they show how to compute intermediate iterates with shorter quantities using two guard digits. For polynomials the situation is simpler and these shorter intermediate results may be used without guard digits.

Algorithm 1 Generic iteration to compute $\mathrm{shinv}_h(v)$ in D given initial approximation w_0

function SHINV (v, h, k, w_0, ℓ)
  ▷ *w is the current approximation. ℓ is the number of leading correct places of w.*
  ▷ *g is the number of guard places. d is the precision doubling shortfall.*
  if HASCARRIES(D) **then** $g \leftarrow 2;\ d \leftarrow 1$ **else** $g \leftarrow 0;\ d \leftarrow 0$
  $w \leftarrow \mathrm{shift}(w,\ g)$
  while $h - k + 1 - d > \ell$ **do**
    $m \leftarrow \min(h - k + 1 - \ell,\ \ell);\quad s \leftarrow \max(0,\ k - 2\ell + 1 - g)$
    $w \leftarrow \mathrm{shift}_{-d}\left(\textsc{Step}\big(k + \ell + m - s - 1 + d + g,\ \mathrm{shift}_{-s}\, v,\ w,\ m,\ \ell - g\big)\right)$
    $\ell \leftarrow \ell + m - d$
  return $\mathrm{shift}_{-g}(w)$

function STEP $(h, v, w, m, \ell) = \mathrm{shift}_m\, w + \mathrm{shift}_{2m-h}\left(w \times \textsc{PowDiff}(v, w, h - m, \ell)\right)$

function POWDIFF (v, w, h, ℓ) ▷ *Compute $\mathrm{shift}_h\, 1 - v \times w$ efficiently.*
  $c \leftarrow$ **if** HASCARRIES(D) **then** 1 **else** 0
  $L \leftarrow \mathrm{prec}\ v + \mathrm{prec}\ w\ - \ell + c$ ▷ *c for coeff to peek*
  if $v = 0 \vee w = 0 \vee L \geq h$ **then return** $\mathrm{shift}_h\, 1 - v \times w$
  else
    $P \leftarrow \mathrm{multmod}(v, w, L)$ ▷ *Lower L places of product vw.*
    if HASCARRIES$(D) \wedge \mathrm{coeff}(P, L - 1) \neq 0$ **then return** $\mathrm{shift}_L\, 1 - P$
    else return $-P$

Theorem 3 (Shift Extension)**.** *Let $w = \mathrm{shinv}_h\, v$, $B^k \leq v < B^{k+1} \leq B^h$ and let $w_{[n]} = \mathrm{shift}_{n\ell-h+k}(w)$ be the leading $n\ell$ digits of w, with $n\ell \leq h - k$. Then*

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

Theorem 4 (Divisor Sensitivity)**.** *Let $w_{[n]}$ be as in Theorem 3 and let Δ be the change obtained by perturbing the divisor v by δ 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

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

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

Theorem 5 (Close Differences)**.** *When $\left| B^h - vw \right| \leq B^e, \quad e < h$, only the lower e digits of the product vw need be computed since the upper $h - e$ digits will be determined. The quantity e satisfies*

$$e \leq k + t - \ell + g,$$

where $\mathrm{prec}\ v = k + 1$ *and* $\mathrm{prec}\ w = t + 1$*, ℓ is the number of known correct places in w and g is the required number of guard digits.*

Theorems 3 and 4 together show two guard digits are required when the domain is $\mathbb{Z}$. None are required for polynomials since there cannot be carries. Theorem 5 shows how to compute the difference $\mathrm{shift}_h\, 1 - vw$, using only a suffix of vw. This will give a savings for some multiplication algorithms, but not for the asymptotically fastest ones.

These results can be extended to non-commutative polynomials with left and right quotients [9]. Define "lquo" and "rquo" by $u = v \times (u \operatorname{lquo} v) + r_L = (u \operatorname{rquo} v) \times v + r_R$, where each of r_L and r_R either zero or with degree less than v. For $R[x]$, where R is not necessarily commutative, shinv is remains well-defined, that is it can be shown $\operatorname{shinv}_{n,x}(v) = x^n \operatorname{lquo} v = x^n \operatorname{rquo} v$. It remains the case that shinv may be computed in a logarithmic number of steps and quotients may be computed according to the following:

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 $h \geq \operatorname{degree} u$,*

$$u \operatorname{lquo} v = \operatorname{shift}_{-h}(\operatorname{shinv}_h(v) \times u) \qquad u \operatorname{rquo} v = \operatorname{shift}_{-h}(u \times \operatorname{shinv}_h(v)). \tag{3}$$

To have a well-defined notion of degree for polynomials where the variable does not commute with the coefficients, we are led to skew polynomials as Ore extensions [2, 4]. In this case, we may define the left (right) whole shift by multiplying on the left (right) by x^n and left (right) whole shifted inverse as the left (right) quotient of x^n and we have the following theorem, though the computation is no longer asymptotically fast.

Theorem 7 (Right quotient from the whole shifted inverse in $R[x; \sigma, \delta]$)**.** *Let $u, v \in R[x, \delta]$, R a ring, $k = \operatorname{degree} v$, and v_k invertible in R. Then, for $h \geq \operatorname{degree} u$,*

$$u \operatorname{rquo} v = \operatorname{rshift}_{-h}(u \times \operatorname{lshinv}_h v). \tag{4}$$

3. Inversion without Pivots for Block Matrices over Division Rings

As has been noted by Abdali and Wise [1], a useful computational representation of matrices over a ring R is with a recursive 2×2 block structure. This representation allows efficient implementation of sub-n^3 multiplication and related operations [6]. It supports row-based and column-based traversal equally well, it is reasonably efficient in representing dense, sparse and structured matrices, and it can provide good locality of reference for block algorithms. Most algebraic operations can be expressed naturally in terms of a recursive abstract data type represented as quadtrees with leaves in R and, if desired, these may be stored densely without using pointers [5].

Most ring operations on block matrices may be performed in a straightforward manner using only block operations. That is, for a block matrix $M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$, only ring operations involving A, B, C and D are required. Computing the matrix inverse is less obvious, however. If all of the blocks of M are invertible, the inverse of M may be computed as

$$M^{-1} = \begin{bmatrix} (A - BD^{-1}C)^{-1} & (C - DB^{-1}A)^{-1} \\ (B - AC^{-1}D)^{-1} & (D - CA^{-1}B)^{-1} \end{bmatrix}.$$

In practice, the usual approach is to compute only two inverses—that of A and that of its Schur complement, $S_A = D - CA^{-1}B$,

$$M^{-1} = \begin{bmatrix} I & -A^{-1}B \\ 0 & I \end{bmatrix} \begin{bmatrix} A^{-1} & 0 \\ 0 & S_A^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -CA^{-1} & I \end{bmatrix} = \begin{bmatrix} A^{-1} + A^{-1}BS_A^{-1}CA^{-1} & -A^{-1}BS_A^{-1} \\ -S_A^{-1}CA^{-1} & S_A^{-1} \end{bmatrix}. \tag{5}$$

If A is not invertible, then a similar formula involving the inverse of another block and its Schur complement may be used, perhaps after a permutation of rows or columns. The problem with this approach is that M may be invertible even when all of A, B, C and D are singular. In this situation, permuting the blocks is of no help. One approach is to break the block abstraction and use operations on whole rows of M viewed as a flat $2^k \times 2^k$ matrix [3].

In earlier work [7], we have shown how to compute inverses using only block operations, without pivots and without breaking the block abstraction. The technique is to use the Moore-Penrose inverse so that the principal minors are guaranteed to be invertible and equation (5) may be used. We summarize those results here. We use the notation $R^{(2\times 2)^k}$ to mean the ring of $2^k \times 2^k$ matrices with elements in R, structured in recursive 2×2 blocks. Any $n \times n$ matrix may be easily be embedded in such a ring.

Theorem 8. *If R is a formally real division ring and $M \in R^{n\times n}$ is invertible, then it is possible to compute M^{-1} as $(M^T M)^{-1} M^T$ using only block operations. By block operations, we mean ring operations in $R^{(2\times 2)^k}$.*

Examples of formally real rings are $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{Q}[\sqrt{2}]$ and $R[x, \partial]$ for formally real R.

Theorem 9. *Let C be a division ring with a formally real sub-ring R and involution "$*$", such that for all $c \in C$, $c^* \times c$ is a sum of squares in R. If $M \in C^{n\times n}$ is invertible, then it is possible to compute M^{-1} as $(M^* M)^{-1} M^*$ using only block operations. Here, block operations are ring operations in $C^{(2\times 2)^k}$.*

Examples of such rings are the complexification of a formally real ring R as $R[i]/\langle i^2 + 1\rangle$ or quaternions over R with the involution $(a + bi + cj + dk)^* = a - bi - cj - dk$.

Theorem 10. *Let K be a field. If $M \in K^{n\times n}$ is invertible, then it is possible to compute M^{-1} as $(M^\circ M)^{-1} M^\circ$ using only block operations, that is ring operations in $K(t)^{(2\times 2)^k}$.*

Here $M^\circ = Q_n^{-1} M^T Q_n$ is a group conjugate of M^T, with $Q_n = \mathrm{diag}(1, t, \ldots, t^{n-1})$.

4. Conclusion

We have shown two generic techniques to compute multiplicative inverses efficiently in algebraic domains of importance to computer algebra. These methods are supported by theorems stated here and proven in earlier work.

References

[1] S. Kamal Abdali and David S. Wise. Experiments with quadtree representation of matrices. In *Symbolic and Algebraic Computation*, pages 96–108. Springer, 1989.

[2] Sergei A. Abramov, H. Q. Le, and Ziming Li. Univariate Ore polynomial rings in computer algebra. *Journal of Mathematical Sciences*, 131(5):5885–5903, 2005.

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

[4] Manuel Bronstein and Marko Petkovšek. An introduction to pseudo-linear algebra. *Theoretical Computer Science*, 157(1):3–33, 1996.

[5] Irene Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–910, December 1982.

[6] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[7] Stephen M. Watt. Pivot-free block matrix inversion. In *Proc. 8th Int'l Symp. on Symbolic and Numeric Algorithms for Scientific Computing*, pages 151–155. IEEE Press, 2006.

[8] Stephen M. Watt. Efficient generic quotients using exact arithmetic. In *Proc. International Symposium on Symbolic and Algebraic Computation*, New York, 2023. ACM.

[9] Stephen M. Watt. Efficient quotients of non-commutative polynomials, arxiv:2305.17877, 2023.