Deterministic Unimodularity Certification and Applications for Integer Matrices

by

Colton Pauderis

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2013

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The asymptotically fastest algorithms for many linear algebra problems on integer matrices, including solving a system of linear equations and computing the determinant, use high-order lifting. For a square nonsingular integer matrix $A \in \mathbb{Z}^{n \times n}$, high-order lifting computes $B \equiv A^{-1} \mod X^k$ and $R \in \mathbb{Z}^{n \times n}$ with $AB = I + RX^k$ for $X, k \in \mathbb{Z}_{\geq 0}$. Here, we present a deterministic method — "double-plus-one" lifting — to compute the high-order residue R as well as a succinct representation of B. As an application, we give a fully deterministic algorithm to certify the unimodularity of $A \in \mathbb{Z}^{n \times n}$. The cost of the algorithm is $O((\log n) n^\omega \, \mathsf{M}(\log n + \log ||A||))$ bit operations, where $||A||$ denotes the largest entry in absolute value, $\mathsf{M}(t)$ the cost of multiplying two integers bounded in bit length by t, and ω the exponent of matrix multiplication.

Unimodularity certification is then applied to give a heuristic, but certified, algorithm for computing the determinant and Hermite normal form of a square, nonsingular integer matrix. Though most effective on random matrices, a highly optimized implementation of the latter algorithm demonstrates the techniques' effectiveness across a variety of inputs: empirical running times grow as $O(n^3 \log n)$. A comparison against the fastest known Hermite normal algorithms — those available in Sage and Magma — show our implementation is, in all cases, highly competitive, and often surpasses existing, state-of-the-art implementations.

Acknowledgements

Primary acknowledgements, thanks, and gratitude are owed to my supervisor, Arne Storjohann, for his seemingly endless fount of ideas, encouragement, advice, and stories; I could not have hoped for a better supervisior. Mark Giesbrecht and George Labahn are due additional thanks, not just for their feedback in the preparation of this document, but for all manner of guidance throughout my time with the Symbolic Computation Group. Of course, I am also indebted to the other graduate students at the SCG (as well as my civilian friends) for reasons too numerous, varied, and specific to list here.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

> Mathematical symbols do not readily lend themselves to the double entendre.
>
> Fran Lebowitz. *Metropolitan Life* (1978).

When you really think about it, that "lifting" algorithms are so-named displays a surprising amount of metaphorical flair and panache; that there are so many different types displays a great fondness for said flair. A lifting algorithm may refer to any one of a loosely-related class of "prime power" algorithms in which some amount of hard work is first done directly at a lower precision and is subsequently used to derive results at successively higher precision: an initial solution is said to be "lifted" to a more accurate one[1]. Here, we focus on the technique of "high-order lifting" for linear algebra: inverting an integer matrix, specifically.

Given a square, nonsingular, integer matrix $A \in \mathbb{Z}^{n \times n}$ along with an integer lifting modulus $X \in \mathbb{Z}_{\geq 2}$, high-order lifting computes a representation of $B \equiv A^{-1} \bmod X^k$ and a high-order residue $R \in \mathbb{Z}^{n \times n}$ with $AB = I + Rx^k$ for some precision $k \in \mathbb{Z}_{\geq 0}$. The standard algorithms of linear X-adic lifting and quadratic lifting (algebraic Newton iteration), as well as the asymptotically fast shifted number system-based method [38] all achieve this goal. While each begins by directly computing $A^{-1} \bmod X$, these algorithms are distinguished from each other by the manner in which they perform the "lifting" and by the representation of B they compute. Though the shifted number system performs high-order

[1] The broad applicability of the metaphor necessitates that a general description be commensurately vague.

lifting in matrix multiplication time (up to logarithmic factors) and its applications to several other linear algebra problems are well described, the algorithm requires randomization and does not lend itself well to a high-performance implementation. As such, we introduce the method of "double-plus-one" lifting to address these concerns.

As the name rather obliquely suggests, double-plus-one lifting alternates between steps of a division-free variant of quadratic lifting and standard linear X-adic lifting. Beginning with the initial inverse $B_0 = A^{-1} \bmod X$, a step of quadratic lifting doubles the precision of the inverse to yield $B_1 \equiv A^{-1} \bmod X^2$. The titular lack of division in division-free quadratic lifting means, in general, B_1 may exceed $(A^{-1} \bmod X^2)$, though modular equivalence always holds. A step of linear lifting increases the precision by a power of X, corrects for any overflow, and yields $B_2 \equiv A^{-1} \bmod X^3$. Double-plus-one lifting also produces a sparse inverse expansion, an expression for $A^{-1} \bmod X^{2^k-1}$ of the form

$$\left(((B_0(I + R_0X) + M_0X^2)(I + R_1X^3) + M_1X^6)\cdots\right)\left((I + R_{k-2}X^{2^{k-1}-1}) + M_{k-2}X^{2^k-2}\right).$$

Each $R_i \in \mathbb{Z}^{n\times n}$ corresponds to a step of quadratic lifting while each $M_i \in \mathbb{Z}^{n\times n}$ corresponds to a step of linear lifting. This expression has logarithmically many terms each of which has bounded entries not much larger than those of the input matrix.

In addition, double-plus-one lifting computes a high-order residue $R \in \mathbb{Z}^{n\times n}$ such that $A(A^{-1} \bmod X^k) = I + RX^k$, or, equivalently, $A^{-1} = (A^{-1} \bmod X^k) + A^{-1}RX^k$. The latter equation inspires a first application of double-plus-one lifting. If $A \in \mathbb{Z}^{n\times n}$ is a unimodular matrix ($\det A = \pm 1$), then A^{-1} is also integral and, for k large enough, $A^{-1} = (A^{-1} \bmod X^k)$. In other words, A^{-1} has a finite X-adic expansion. If this is indeed the case, the high-order residue R must be the zero matrix. Moreover, $R = 0$ is a necessary and sufficient condition for A to be unimodular, provided X and k are large enough. Making these requirements precise yields a fully deterministic method for unimodularity certification.

Briefly deferring the issue of the utility of unimodularity certification, we next develop an algorithm for the determinant and Hermite normal form[2] of a square, nonsingular, integer matrix $A \in \mathbb{Z}^{n\times n}$. We rely on the observation that the rational solution $X \in \mathbb{Q}^{n\times 1}$ to the linear system $Ax = v$ for a random vector $v \in \mathbb{Z}^{n\times 1}$ reveals information about the determinant. Specifically, the lowest common multiple of the denominators of x will be some divisor of the determinant of A, and quite likely a large divisor. Two issues immediately arise. Firstly, once a single system solution provides a single divisor of the

[2]The Hermite normal form is a canonical triangular representation of the lattice generated by the rows of the matrix; the details can safely be ignored at this juncture.

determinant, it is not clear how to obtain additional divisors, perhaps by modifying A. We solve this problem by giving an efficient algorithm to compute a so-called minimal triangular denominator of a rational solution vector x: an upper triangular matrix T that clears the denominator of x (i.e., Tx is integral)[3]. We may then work with $(AT^{-1}) \in \mathbb{Z}^{n \times n}$ to obtain further divisors of $\det A$, iterating as needed. This leads to the second issue: as the linear solving method for extracting factors of $\det A$ is probabilistic, some reliable termination condition is desired. Almost too conveniently, deterministic unimodularity certification provides such a check: if the work matrix is unimodular, the entirety of the determinant must have been extracted. This yields a certified algorithm.

Returning to the initial claim that the shifted number system method of high-order lifting was unsuitable for implementation, we detail an optimized implementation of double-plus-one lifting, unimodularity certification, and the consequent algorithm for Hermite normal form. The implementation makes use of a residue number system to allow computations to be performed with native, word-size arithmetic routines and to take advantage of practically fast matrix multiplication whenever possible. As a rather satisfying conclusion, empirical timings show our implementation for integer Hermite normal form is competitive with and often surpasses the fastest known implementations: those available in computer algebra systems Sage and Magma.

1.1 Preliminaries

Cost estimates are occasionally given in terms of one of more cost functions for a few common operations, abstracting away the complexity specific to any one realization. The cost of multiplying two t-bit integers (integers bounded in magnitude by 2^t) is denoted $\mathsf{M}(t)$. Naïve methods give $\mathsf{M}(t) \in O(t^2)$ while more advanced techniques give $\mathsf{M}(t) \in O(t(\log t)(\log\log t))$. Likewise, $\mathsf{B}(t)$ denotes the cost of the extended Euclidean algorithm (compute a greatest common divisor and Bézout coefficients) with two t-bit integers. Again, depending on the choice of technique, $\mathsf{B}(t) \in O(t^2)$ or $\mathsf{B}(t) \in O(\mathsf{M}(t)\log t)$. Finally, ω denotes the exponent of matrix multiplication with $2 \leq \omega \leq 3$: the product of two $n \times n$ matrices over a commutative ring (but always the integers, for our purposes) may be computed in at most $O(n^\omega)$ ring operations. For more details about integer and matrix multiplication we refer to the textbook by von zur Gathen and Gerhard [11].

Let $X > 2$ be an integer. For any rational number a that has denominator relatively prime to X, we let $\mathrm{Rem}(a, X)$ denote the unique integer in the usual "symmetric range"

[3] T also has the property of revealing the portion of the Hermite form corresponding to the particular divisor of $\det A$ at hand.

modulo X, that is, $\mathrm{Rem}(a, X) \equiv a \bmod X$ and

$$\mathrm{Rem}(a, X) \in \left[-\left\lfloor \frac{X-1}{2} \right\rfloor, \left\lfloor \frac{X}{2} \right\rfloor \right].$$

The next two lemmas follow directly from the above definition.

Lemma 1. $|\mathrm{Rem}(*, X)|/X \leq 1/2$.

Lemma 2. *If* $a \in \mathbb{Z}$ *satisfies* $|a| < X/2$ *then* $\mathrm{Rem}(a, X) = a$.

For a matrix A of rational numbers having denominator relatively prime to X, the quantity $\mathrm{Rem}(A, X)$ denotes the matrix obtained from A by applying $\mathrm{Rem}(*, X)$ to each entry.

Throughout, $||A|| = \max |A_{ij}|$ denotes the maximum magnitude of entries in A.

Hadamard's inequality will be a useful tool as it gives a bound for the determinant of A: $|detA| \leq n^{n/2}||A||^n$ [11, Theorem 16.6].

Many complexity results are stated in "soft-oh" notation to suppress logarithmic factors in variables appearing elsewhere: $f(n) \in O\tilde{}(g(n))$ means $f(n) \in O(g(n) \log^k g(n))$ for some positive integer k.

Earlier versions of these results appear in [28, 29]. Double-plus-one lifting and the consequent algorithm for deterministic unimodularity certification are developed in Chapter 2. Chapter 3 further extends these techniques to give heuristic, but certified algorithms for the determinant and Hermite normal form. An optimized C implementation is described in Chapter 4, along with empirical timings and implementation-specific tricks and traps. Chapter 5 is the last chapter.

Chapter 2

Double-Plus-One Lifting and Deterministic Unimodularity Certification

> The rain man gave me two cures then he said, "Jump right in."
> The one was Texas medicine; the other was just railroad gin.
> And like a fool I mixed them and it strangled up my mind.
>
> Bob Dylan. "Stuck Inside of Mobile with the Memphis Blues Again", *Blonde on Blonde* (1966).

Combining the standard methods of linear and quadratic X-adic lifting, the central result of this chapter is "double-plus-one" lifting, an asymptotically fast method for computing a sparse representation of $B = A^{-1} \bmod X^k$ and a high-order residue $R \in \mathbb{Z}^{n \times n}$ with $AB = I + RX^k$ for $k \in \mathbb{Z}^{\geq 0}$. Moreover, double-plus-one lifting works in the symmetric range modulo X and avoids the randomized number system required by the best previous method for this type of high-order lifting. A fully deterministic algorithm to assay if an $n \times n$ integer matrix A is unimodular is presented as an application. The cost of the algorithm is $O((\log n) n^\omega \mathsf{M}(\log n + \log ||A||))$. The results of this chapter appeared previously in [28].

2.1 Introduction

Although "lifting" as a problem solving framework may refer to several techniques in computer algebra, the modular, prime power-based central conceit is essentially the same. Initially, some otherwise hard work is done once modulo some preferably small prime (prime in some appropriate problem-specific sense), the *lifting modulus*. This hard work is generally related to the problem at hand — perhaps factoring a polynomial (as in Hensel lifting), polynomial division (Newton iteration), or solving a linear system (of interest here) — and is made tractable by the requirement that the solution be accurate only in a modular sense.

All the lifting algorithms discussed herein operate on a square, nonsingular integer matrix $A \in \mathbb{Z}^{n\times n}$ and a lifting modulus $X \in \mathbb{Z}$. The lifting modulus X is additionally required to be relatively prime to the determinant of A. For now, we assume a suitable modulus is in-hand and briefly defer discussion as to its acquisition. This initial solution is then extended to yield a solution accurate to successively higher powers of the lifting modulus. In the metaphorical parlance that gives the technique its name, we say the initial solution is "lifted" to a higher precision. Often, but not always, each subsequently more precise solution depends on only the previous solution, and not the entire sequence.

For A and X as above, lifting computes $R \in \mathbb{Z}^{n\times n}$ and $B \in \mathbb{Z}^{n\times n}$ satisfying the equation

$$A = B + A^{-1}RX^k$$

for a non-negative integer k. We call matrix R the *residue* and integer k the *precision* of the lifting. Note also that $B \in \mathbb{Z}^{n\times n}$ is necessarily congruent to $A^{-1} \bmod X^k$. Depending on the application and the algorithm in play, only one of R and B may be required.

An additional important aspect of these lifting algorithms is their ability to compute a useful implicit representation of B rather than explicitly computing B itself. As the precision k increases, so to does the total size of $B \equiv A^{-1} \bmod X^k$. In many cases, the size of B quickly exceeds that at which B is of any practical use.

For instance, high-order lifting by way of the shifted number system [38] requires that the modulus satisfies $\log X \in \Theta(\log n + \log ||A||)$ and the precision satisfies $k \in O(n)$. In this framework, then, writing down B explicitly requires $\Omega(n^2 \log(X^k)) = \Omega(n^3(\log n + \log ||A||))$ bits of space. To avoid having to work with such a large quantity, high-order lifting computes a *sparse inverse expansion* of the form

$$A = \overbrace{((\cdots(*(I + *X^2) + *X^4)(I + *X^4) + \cdots) + *X^k)}^{B} + A^{-1}RX^k.$$

This expansion has two important properties. First, there are a logarithmic number (in k) of terms in the expansion. Moreover, each term is bounded in magnitude: $||*|| \in O(n||A||)$. The combination of these two factors gives that the total size of the expansion is $O(n^2 \log n(\log n + \log ||A||)$ bits. Note that, compared to the explicit representation, the sparse representation reduces a linear term of n to a logarithmic one.

The asymptotically fastest algorithms for many linear algebra problems on integer matrices, including solving a system of linear equations and computing the determinant, use high-order lifting. For computing $A^{-1}b$, the p-adic lifting algorithm of Dixon [9] already achieves an expected running time of $O\tilde{}(n^3 \log ||A||)$ bit operations. High-order lifting in a shifted number system [38] allows the incorporation of matrix multiplication to reduce the expected running time to $O\tilde{}(n^\omega \log ||A||)$. The exponent of n improves from 3 to ω.

For other problems, however, the improvement in running time is more fundamental than simply permitting fast matrix multiplication to be fully leveraged. For example, without the use of sub-cubic matrix multiplication, the previously fastest algorithm for computing $\det A$ uses an expected number of $O\tilde{}(n^{3.2} \log ||A||)$ bit operations and $O\tilde{}(n^{2.2} \log ||A||)$ bits of intermediate space [22]. The algorithm for $\det A$ based on high-order lifting [38, §13] uses an expected number of $O\tilde{}(n^3 \log ||A||)$ bit operations and is space-efficient: intermediate space requirements are bounded by $O\tilde{}(n^2 \log ||A||)$ bits. The problem of integrality certification — determining if $A^{-1}C$ is integral for a second given matrix C — provides an additional example of the utility of high-order lifting. If $||C|| \in \Theta(||A||)$, then directly computing $A^{-1}C$ using a standard method such as quadratic lifting or Chinese remainder theorem-based modular imaging would require on the order of $O\tilde{}(n^{\omega+1} \log ||A||)$ bit operations and $\Omega(n^3 \log ||A||)$ bits of intermediate space. High-order lifting can answer the integrality certification question with a space-efficient algorithm in a Las Vegas fashion using an expected number of $O\tilde{}(n^\omega \log ||A||)$ bit operations [38, §11]. Here, in both time and space complexities, the exponent of n is decreased by one.

High-order lifting requires a modulus X that is relatively prime to $\det A$. If a suitable modulus is not known *a priori*, one can be constructed as the power of a prime p that is randomly chosen in the range $2^{\ell-1} < p < 2^\ell$, where $\ell = 6 + \ln \ln(\delta)$ and $\delta = n^{n/2}||A||^n$. As $|\det A|$ is at most δ (Hadamard's inequality), fewer than half of the primes p selected from this range can possibly divide $\det A$ [12, Theorem 1.8]. For the most part, we politely ignore this technicality and assume a suitable modulus is given.

The remainder of this chapter describes the well-known linear and quadratic lifting algorithms before introducing "double-plus-one" lifting as a synthesis of both addressing the limitations of each. A fully deterministic method for unimodularity certification is given as an application of double-plus-one lifting.

2.2 X-adic lifting

Throughout this section, $A \in \mathbb{Z}^{n\times n}$ is nonsingular and $X \in \mathbb{Z}_{>2}$ is relatively prime to $\det A$; both are assumed to be given. Beginning with a local inverse $\operatorname{Rem}(A^{-1}, X)$, X-adic lifting increases the precision to obtain $\operatorname{Rem}(A^{-1}, X^k)$. This section describes the standard linear and quadratic X-adic lifting algorithms in §2.2.1 and §2.2.2, respectively; in §2.2.3, we give a division-free variant of quadratic lifting that yields a straight line formula for $B \equiv \operatorname{Rem}(A^{-1}, X^k) \bmod X^k$. These standard techniques form the basis for the double-plus-one lifting algorithm of the following section. We illustrate these concepts with a recurring example using input matrix $A = \begin{bmatrix} 47 & 31 \\ 29 & 74 \end{bmatrix}$ with $\det A = 2579$ and lifting modulus $X = 100$.

2.2.1 Linear lifting

As the name suggests, linear lifting computes an expression for which the computed truncation increases in precision by a single power of X at each iteration. Linear X-adic lifting is based on the identity

$$A^{-1} = \overbrace{C_0 + C_1X + \cdots + C_{i-1}X^{i-1}}^{\operatorname{Rem}(A^{-1}, X^i)} + A^{-1}R_iX^i. \tag{2.1}$$

Rearranging the above equation gives that the residue R_i is equal to

$$R_i = (1/X^i)(I - A\operatorname{Rem}(A^{-1}, X^i)). \tag{2.2}$$

Referring to the example for concreteness, the central linear lifting identity looks as follows (again, with modulus $X = 100$).

$$\overbrace{\begin{bmatrix} 47 & 31 \\ 29 & 74 \end{bmatrix}^{-1}}^{A^{-1}} = \overbrace{\begin{bmatrix} 6 & 11 \\ 49 & 93 \end{bmatrix} + \begin{bmatrix} 74 & 4 \\ 84 & 38 \end{bmatrix} X + \begin{bmatrix} 10 & 86 \\ 99 & 24 \end{bmatrix} X^2}^{\operatorname{Rem}(A^{-1}, X^3)} + A^{-1}\overbrace{\begin{bmatrix} -36 & -48 \\ -77 & -43 \end{bmatrix}}^{R_3} X^3$$

In particular, note that the truncation of the inverse, $\operatorname{Rem}(A^{-1}, X^i)$, is expressed in

terms of its X-adic expansion $C_0, C_1, C_2, \ldots, C_{i-1}$.

$$\overbrace{\begin{bmatrix} 47 & 31 \\ 29 & 74 \end{bmatrix}^{-1}}^{A^{-1}} \equiv \begin{bmatrix} 107406 & 860411 \\ 998449 & 243893 \end{bmatrix} \bmod X^3$$
$$= \overbrace{\begin{bmatrix} 6 & 11 \\ 49 & 93 \end{bmatrix}}^{C_0} + \overbrace{\begin{bmatrix} 74 & 4 \\ 84 & 38 \end{bmatrix}}^{C_1} X + \overbrace{\begin{bmatrix} 10 & 86 \\ 99 & 24 \end{bmatrix}}^{C_2} X^2$$

As is typical of lifting algorithms, each iterate C_i and R_{i+1} in the sequence can be computed from the iterates immediately previous: earlier coefficients in the X-adic expansion are not required. At iteration $i-1$, coefficient C_{i-1} and residue R_i are known. From these quantities, as well as the input A and the initial truncation C_0, the next iterates C_i and R_{i+1} can be then computed. Specifically, the next X-adic coefficient C_i is computed from the previous residue R_i as

$$C_i = \mathrm{Rem}(C_0 R_i, X).$$

Likewise, the next residue R_{i+1} is computed from the previous X-adic coefficient C_i as

$$R_{i+1} = (1/X)(R_i - AC_i). \tag{2.3}$$

For instance, if C_1 and R_2 are known as below,

$$\overbrace{\begin{bmatrix} 47 & 31 \\ 29 & 74 \end{bmatrix}^{-1}}^{A^{-1}} = \overbrace{\begin{bmatrix} 6 & 11 \\ 49 & 93 \end{bmatrix}}^{C_0} + \overbrace{\begin{bmatrix} 74 & 4 \\ 84 & 38 \end{bmatrix}}^{C_1} X + A^{-1} \overbrace{\begin{bmatrix} -61 & -14 \\ -84 & -30 \end{bmatrix}}^{R_2} X^2,$$

C_2 and R_3 can be computed as

$$C_2 = C_0 R_2 = \begin{bmatrix} 10 & -14 \\ -1 & 24 \end{bmatrix} R_3 \qquad = \frac{R_2 - AC_2}{X} = \begin{bmatrix} -36 & -48 \\ 23 & -43 \end{bmatrix}$$

and the X-adic expansion can be extended by a single power of X:

$$\overbrace{\begin{bmatrix} 47 & 31 \\ 29 & 74 \end{bmatrix}^{-1}}^{A^{-1}} = \overbrace{\begin{bmatrix} 6 & 11 \\ 49 & 93 \end{bmatrix}}^{C_0} + \overbrace{\begin{bmatrix} 74 & 4 \\ 84 & 38 \end{bmatrix}}^{C_1} X + \overbrace{\begin{bmatrix} 10 & 86 \\ 99 & 24 \end{bmatrix}}^{C_2} X^2 + A^{-1} \overbrace{\begin{bmatrix} -36 & -48 \\ -77 & -43 \end{bmatrix}}^{R_3} X^3.$$

Algorithm 1 `linearLift`(A, X, k)

Input: $A \in \mathbb{Z}^{n\times n}$, $X \in \mathbb{Z}$ with $X \perp \det A$, $k \in \mathbb{Z}_{>0}$.
Output: $C_0 \ldots, C_{k-1}, R_k \in \mathbb{Z}^{n\times n}$ as in (2.1).

1: $C_0 := \mathrm{Rem}(A^{-1}, X)$
2: $R_1 := (1/X)(I - AC_0)$
3: **for** $i = 1$ **to** $k - 1$ **do**
4: $\quad C_i := \mathrm{Rem}(C_0 R_i, X)$
5: $\quad R_{i+1} := (1/X)(R_i - AC_i)$
6: **end for**
7: **return** $C_0, \ldots, C_{k-1}, R_1, \ldots, R_k$

Algorithm 1 summarizes this process and gives the standard algorithm to compute the X-adic expansion of A^{-1}. Again, note that C_i and R_{i+1} can be computed using only A, R_i and C_0: the other coefficients of the X-adic expansion of A^{-1} are not required. The following theorem captures this essential idea of linear X-adic lifting.

Theorem 3. *Let $A \in \mathbb{Z}^{n\times n}$ be nonsingular and $X \in \mathbb{Z}_{>2}$ be relatively prime to $\det A$. If $B, R \in \mathbb{Z}^{n\times n}$ satisfy*

- $A^{-1} = B + A^{-1}RX^k$

for some $k > 0$, then for any $M \in \mathbb{Z}^{n\times n}$ such that $M \equiv A^{-1}R \bmod X^\ell$ for some $\ell > 0$, we have

- $A^{-1} = B + MX^k + A^{-1}R'X^{k+\ell}$,

where $R' := (1/X^\ell)(R - AM)$.

Proof. As $M \equiv A^{-1}R \bmod X^\ell$, $AM \equiv R \bmod X^\ell$ and, equivalently, $AM = R - R'X^\ell$, for some $R \in \mathbb{Z}^{n\times n}$. Thus, $(1/X^\ell)(R - AM)$ is integral.

Moreover, $A^{-1} \equiv B + MX^k \bmod X^{k+\ell}$ as

$$A(B + MX^k) = (I - RX^k) + (RX^k - R'X^{k+\ell}) \equiv I \bmod X^{k+\ell}.$$

□

In particular, we can choose $\ell = 1$ and $M := \mathrm{Rem}(A^{-1}R, X)$.

Corollary 4. *Let $A \in \mathbb{Z}^{n\times n}$ be nonsingular and $X \in \mathbb{Z}_{>2}$ be relatively prime to* $\det A$. *If $B, R \in \mathbb{Z}^{n\times n}$ satisfy*

- $A^{-1} = B + A^{-1}RX^k$

for some $k > 0$, then for any $M \in \mathbb{Z}^{n\times n}$ such that $M \equiv A^{-1}R \bmod X$ setting $R' := (1/X)(R - AM)$ and $B' := B + MX^k$ gives

- $A^{-1} = B' + A^{-1}R'X^{k+1}$.

We note that the resulting X-adic expression has each term bounded by X. However, the expression itself contains k (i.e., a linear number) of these terms. If $k \in O(n)$ and $\log X \in O(\log n + \log ||A||)$, the total size of the X-adic expansion is $n^3(\log n + \log ||A||)$ bits - no smaller than explicitly representing the inverse itself.

2.2.2 Quadratic lifting

While linear lifting increases the precision of the inverse by a single power of X at each lifting step, quadratic lifting, also known as algebraic Newton iteration, doubles the precision at each step.

Quadratic lifting maintains the identity

$$A^{-1} = \operatorname{Rem}(A^{-1}, X^{2^i}) + A^{-1}RX^{2^i}, \tag{2.4}$$

where the residue R is equal to

$$R = (1/X^{2^i})(I - A\operatorname{Rem}(A^{-1}, X^{2^i})). \tag{2.5}$$

In essence, the quadratic lifting equations (2.4) and (2.5) correspond to the equations underlying linear lifting((2.1) and (2.2)) with the precision i replaced by 2^i. This change reflects the move from an precision-*incrementing* process to a precision-*doubling* one.

Linear lifting uses the initial inverse $C_0 = \operatorname{Rem}(A^{-1}, X)$ to increase the precision of the inverse by one at each step. Whereas, quadratic lifting doubles the precision of the inverse at each step using the observation

$$A^{-1} \equiv \operatorname{Rem}(A^{-1}, X^{2^i})(I + RX^{2^i}) \bmod X^{2^{i+1}}. \tag{2.6}$$

For example, if

$$\overbrace{\begin{bmatrix} 47 & 31 \\ 29 & 74 \end{bmatrix}^{-1}}^{A^{-1}} = \overbrace{\begin{bmatrix} 7406 & 411 \\ 8449 & 3893 \end{bmatrix}}^{\mathrm{Rem}(A^{-1},X^2)} + A^{-1} \overbrace{\begin{bmatrix} -41 & -34 \\ -67 & -48 \end{bmatrix}}^{R} X^2$$

then applying (2.6) to compute the new truncated inverse $\mathrm{Rem}(A^{-1}, X^4)$ followed applying by (2.5) to compute the next residue R' yields

$$A^{-1} = \overbrace{\begin{bmatrix} 37107406 & 39860411 \\ 75998449 & 49243893 \end{bmatrix}}^{\mathrm{Rem}(A^{-1},X^4)} + A^{-1} \overbrace{\begin{bmatrix} -30 & -28 \\ -65 & -46 \end{bmatrix}}^{R'} X^4.$$

Repeated application of this process beginning with $\mathrm{Rem}(A^{-1}, X)$ gives Algorithm 2, the standard algorithm ([3, Algorithm 3.1], for one) for computing $\mathrm{Rem}(A^{-1}, X^{2^k})$. The following theorem conveys the central idea of standard quadratic lifting and is analogous to Theorem 3 for linear lifting.

Algorithm 2 `quadraticLift`(A, X, k)

Input: $A \in \mathbb{Z}^{n\times n}$, $X \in \mathbb{Z}$ with $X \perp \det A$, $k \in \mathbb{Z}_{>0}$.
Output: $B = \mathrm{Rem}(A^{-1}, X^{2^k})$

1: $B := \mathrm{Rem}(A^{-1}, X)$
2: **for** $i = 0$ **to** $k - 1$ **do**
3: $\quad R := (1/X^{2^i})(I - AB)$
4: $\quad B := \mathrm{Rem}\left(B(I + RX^{2^i}), X^{2^{i+1}}\right)$
5: **end for**
6: **return** B

Theorem 5. *Let $A \in \mathbb{Z}^{n\times n}$ be nonsingular and $X \in \mathbb{Z}_{>2}$ be relatively prime to $\det A$. If $B, R \in \mathbb{Z}^{n\times n}$ satisfy*

- $A^{-1} = B + A^{-1}RX^k$

then

- $A^{-1} \equiv \overline{B} \bmod X^{2k}$

where $\overline{B} = \mathrm{Rem}(B(I + RX^k), X^{2k})$.

Proof. Consider $A\overline{B} \bmod X^{2k}$.

$$\begin{aligned}
A\overline{B} &\equiv AB(I + RX^k) \bmod X^{2k} \\
&= AB(2I - AB) && \text{since } R = (I - AB)/X^k \\
&= (I - RX^k)(I + RX^k) && \text{since } AB = I - RX^k \\
&= I + R^2X^{2k} \\
&\equiv I \bmod X^{2k} \\
\overline{B} &\equiv A^{-1} \bmod X^{2k}.
\end{aligned}$$

□

Unlike linear lifting, standard quadratic lifting requires the entirety of the computed inverse $B = \mathrm{Rem}(A^{-1}, X^{2^i})$ to produce either the next iterate $\mathrm{Rem}(A^{-1}X^{2^{i+1}})$ or the next residue. In contrast, linear lifting requires only the previous X-adic coefficient to compute the next coefficient or residue. This property also precludes standard quadratic lifting from admitting a useful alternative representation for the inverse: since the entirety of the truncated inverse must be known at each step, producing, in addition an implicit representation is of questionable utility.

Considering the goal of computing a residue R corresponding to a potentially high-precision truncation B, quadratic lifting does address the potentially large number of linear lifting iterations required. However, the necessity of maintaining an explicit inverse means quadratic lifting is not ideal.

2.2.3 Division-free quadratic lifting

A surprisingly simple modification to the standard quadratic lifting algorithm addresses some of these issues and will eventually lead to the synthetic lifting algorithm of the next section.

Suppose $\mathrm{Rem}(A^{-1}, X^{2^i})$ is not known exactly, but only known modulo X^{2^i}: say

$$\overline{B} = \mathrm{Rem}(A^{-1}, X^{2^i}) + CX^{2^i}$$

for some overflow matrix $C \in \mathbb{Z}^{n\times n}$. With standard quadratic lifting, $\mathrm{Rem}(A^{-1}, X^{2^i})$ is explicitly truncated and is known exactly, with no such overflow term. Even in the presence of overflow, however, the identities necessary for quadratic lifting still hold.

If we compute the residue $\overline{R}$ from $\overline{B}$ we obtain

$$\begin{aligned}
\overline{R} &= (1/X^{2^i})(I - A\overline{B}) \\
&= (1/X^{2^i})(I - A\ \mathrm{Rem}(A^{-1}, X^{2^i}) - ACX^{2^i})) \\
&= (1/X^{2^i})(I - (I + RX^{2^i}) - ACX^{2^i})) \\
&= R - AC.
\end{aligned}$$

In the above, we make use of the fact that $A\ \mathrm{Rem}\left(A^{-1}, X^{2^i}\right) = I - RX^{2^i}$, for $R \in \mathbb{Z}^{n\times n}$. Not coincidentally this R corresponds to the residue computed from $\mathrm{Rem}(A^{-1}, X^{2^i})$ as in (2.5); equivalently, R satisfies $A^{-1} = \mathrm{Rem}(A^{-1}, X^{2^i}) + A^{-1}RX^{2^i}$ as in (2.4).

Thus, $\overline{B}$ and $\overline{R}$, despite the overflow term, satisfy the related equation

$$A^{-1} = \overline{B} + A^{-1}\overline{R}X^{2^i} \tag{2.7}$$

since

$$\begin{aligned}
\overline{B} + A^{-1}\overline{R}X^{2^i} &= \mathrm{Rem}(A^{-1}, X^{2^i}) + CX^{2^i} + A^{-1}(R - AC)X^{2^i} \\
&= \mathrm{Rem}(A^{-1}, X^{2^i}) + A^{-1}RX^{2^i} \\
&= A^{-1} \qquad \text{(by equation 2.5)}.
\end{aligned}$$

Although $\overline{B}$ contains the extra overflow term CX^k, this is cancelled out in (2.7) by the negation of the same term appearing in $A^{-1}\overline{R}X^k$.

The above consideration suggests the following modification to an excerpt of Algorithm 2 (`quadraticLift`) to compute $B_i \equiv \mathrm{Rem}(A^{-1}, X^{2^i}) \mod X^{2^i}$ for $i = 0, 1, \ldots, k$.

$B_0 := \mathrm{Rem}(A^{-1}, X)$;
for $i = 0$ to $k - 1$ **do**
  $R_i := (1/X^{2^i})(I - AB_i)$;
  $B_{i+1} := B_i(I + R_iX^{2^i})$
od

The above excerpt differs from the standard presentation of Newton iteration in that the computation of the B_i for $i \geq 1$ does not include a remainder operation to ensure that $B_i = \mathrm{Rem}(A^{-1}, X^{2^i})$.

As presented above, the computation of R_k still requires the computation of each B_i for $i < k$. Avoiding the remainder operation, however, allows the computation of R_k to be dramatically simplified.

$$\begin{aligned} R_i &= (1/X^{2^i})(I - AB_i) \\ &= (1/X^{2^i})(I - AB_{i-1}(I + R_{i-1}X^{2^{i-1}})) \\ &= (1/X^{2^i})(I - (I - R_{i-1}X^{2^{i-1}})(I + R_{i-1}X^{2^{i-1}})) \\ &= R_{i-1}^2. \end{aligned}$$

Given this fact, each subsequent residue can be computed simply by the squaring the previous, without the need to work with the intermediate values B_i. This simplified version of division-free quadratic lifting is shown in Algorithm 3.

Algorithm 3 `divisionFreeQuadraticLift`(A, X, k)

Input: $A \in \mathbb{Z}^{n\times n}$, $X \in \mathbb{Z}$ with $X \perp \det A$, $k \in \mathbb{Z}_{>0}$.
Output: B_0, R_i as in (2.8)

1: $B_0 := \mathrm{Rem}(A^{-1}, X)$
2: $R_0 := (1/X)(I - AB_0)$
3: **for** $i = 1$ **to** $k - 1$ **do**
4: $\quad R_i := R_{i-1}^2$
5: **end for**
6: **return** $B_0, R_0, R_1, \ldots, R_{k-1}$

Once the residues $R_1, R_2, \ldots, R_{k-1}$ have been computed by the simplified division-free quadratic lifting algorithm, $B_k \equiv \mathrm{Rem}(A^{-1}, X^{2^k}) \bmod X^{2^k}$ can be expressed as a straight line formula

$$B_k = B_0(I + R_0X)(I + R_1X^2)(I + R_2X^4)\cdots(I + R_{k-1}X^{2^{k-1}}). \tag{2.8}$$

In terms of the recurring example,

$$\begin{bmatrix} 47 & 31 \\ 29 & 74 \end{bmatrix}^{-1} \equiv \begin{bmatrix} 6 & 11 \\ 49 & 93 \end{bmatrix} \left(I + \begin{bmatrix} -18 & -34 \\ -38 & -72 \end{bmatrix} X\right) \left(I + \begin{bmatrix} 1616 & 3060 \\ 3420 & 6476 \end{bmatrix} X^2\right) \bmod X^4$$

$$\begin{bmatrix} 47 & 31 \\ 29 & 74 \end{bmatrix}^{-1} = \overbrace{\begin{bmatrix} -4255862892594 & -8058760139589 \\ -35730324001551 & -67657750756107 \end{bmatrix}}^{B_2} + A^{-1} \overbrace{\begin{bmatrix} 13076656 & 24761520 \\ 27674640 & 52403776 \end{bmatrix}}^{R_2} X^4.$$

Although simply removing the modular reduction simplified the algorithm while maintaining correctness, it is not without consequence. Rather, the division-free lifting variant has the conspicuously obvious property that both the residues R_i and coefficients B_i grow rapidly and without bound.

Since $B_0 = \mathrm{Rem}(A^{-1}, X)$ and $AB_0 = I + RX$, the initial residue is bounded by $||R_0|| \leq n||A||$. However, $R_{i+1} = R_i^2$ and (again, very obviously) each subsequent residue doubles in bit length; in turn, the overflow of each B_i increases as well.

The following example shows that, even after only two lifting steps, the size of B_i can become quite large.

$$\begin{aligned} B_2 &= B_0(I + R_0X)(I + R_1X^2) \\ &= \begin{bmatrix} -4255862892594 & -8058760139589 \\ -35730324001551 & -67657750756107 \end{bmatrix} \\ &= \overbrace{\begin{bmatrix} 37107406 & 39860411 \\ -24001551 & 49243893 \end{bmatrix}}^{\mathrm{Rem}(A^{-1}, X^4)} + \overbrace{\begin{bmatrix} -42559 & -80588 \\ -357303 & -676578 \end{bmatrix}}^{\text{overflow}} X^4. \end{aligned}$$

We end this section with the following theorem which captures the essential idea of division-free quadratic X-adic lifting.

Theorem 6. *Let $A \in \mathbb{Z}^{n\times n}$ be nonsingular and $X \in \mathbb{Z}_{>2}$ be relatively prime to* $\det A$. *If $B, R \in \mathbb{Z}^{n\times n}$ satisfy*

- $A^{-1} = B + A^{-1}RX$,

then

- $A^{-1} = B(I + RX) + A^{-1}R^2X^2$.

Proof. Multiplying both sides of the equation $A^{-1} = B + A^{-1}RX$ on the right by RX gives an equation for $A^{-1}RX$, $A^{-1}RX = BRX + A^{-1}R^2X^2$. Applying this identity to the equation assumed in the precondition yields the desired result:

$$\begin{aligned} A^{-1} &= B + A^{-1}RX \\ &= B + BRX + A^{-1}R^2X^2. \end{aligned}$$

□

2.3 Double-plus-one lifting

Neither linear nor straight-line quadratic lifting is ideal for the purposes of computing a residue R of high-order. While linear lifting computes small coefficients at each step, the total number of steps is large. Conversely, while straight-line quadratic lifting requires only logarithmically many steps to reach precision X^k, each subsequent residue is twice the size of the last. Considering only number and size of coefficients, the strength of each approach is precisely the weakness of the other. A simple synthesis — in fact, simply an interleaving — of linear and quadratic lifting brings the strength of each to bear on the weakness of the other without, perhaps surprisingly, compromising the advantages of either.

Beginning with a step of quadratic lifting, "double-plus-one" lifting alternates between steps of linear and quadratic X-adic lifting. Consequently, at each lifting stage, the precision of the computed inverse B_i is doubled, plus an additional power of X. That is, B_i congruent to A^{-1} modulo X^k is lifted to yield B_{i+1} congruent to A^{-1} modulo X^{2k+1}.

For convenience, define $X_i = X^{2^{i+1}-1}$. Then $X_{i+1} = X_i^2 X$ for all $i \geq 0$.

$$\begin{aligned} X_0 &= X \\ X_1 &= X_0^2 X = X^{2^2-1} \\ X_2 &= X_1^2 X = X^{2^3-1} \\ &\vdots \\ X_{k-1} &= X_{k-2}^2 X = X^{2^k-1} \end{aligned}$$

More precisely, we initialize $B_0 := \mathrm{Rem}(A^{-1}, X_0 = X)$ and compute, for $i = 0, 1, 2, \ldots$ in succession, the following:

$$\begin{aligned} B_1 &= B_0(I + R_0 X_0) + M_0 X_0^2 \equiv A^{-1} \bmod X_1 (= X_0^2 X) \\ B_2 &= B_1(I + R_1 X_1) + M_1 X_1^2 \equiv A^{-1} \bmod X_2 (= X_1^2 X) \\ &\vdots \end{aligned} \tag{2.9}$$

Each multiplicative factor $(I + R_i X_i)$ encodes a step of quadratic lifting applied to B_i. This doubles the precision to yield $B_i(I + R_i X_i) \equiv A^{-1} \bmod X_{i-1}^2$. Likewise, each additive term $M_i X_i^2$ encodes a step of linear lifting applied to $B_i(I + R_i X_i)$. This increases the precision by a single power of X so that $B_i(I + R_i X_i) + M_i X_i^2 \equiv A^{-1} \bmod X_{i-1}^2 X$.

The resulting expansion — the sparse inverse expansion — has the following form

$$A^{-1} = (\cdots((B_0(I + R_0 X_0) + M_0 X_0^2)(I + R_1 X_1) + M_1 X_1^2) + \cdots).$$

The sparse inverse expansion has a logarithmic number of terms (like quadratic lifting) and each is of bounded size (like linear lifting). Broadly, the overflow incurred by each step of quadratic lifting is accounted for by the subsequent step of linear lifting. Additionally, each R_i and M_i relies only on the previous coefficients, and not the entirety of the expansion.

The approach is best illustrated with a concrete example. As before, let $X = 100$ and $A = \begin{bmatrix} 47 & 31 \\ 29 & 74 \end{bmatrix}$, and consider the computation of A^{-1}. To begin, we initialize $B_0 = \mathrm{Rem}(A^{-1}, X)$ and $R_0 = (1/X)(I - AB_0)$ to obtain

$$\overbrace{\begin{bmatrix} 47 & 31 \\ 29 & 74 \end{bmatrix}^{-1}}^{A^{-1}} = \overbrace{\begin{bmatrix} 6 & 11 \\ 49 & 93 \end{bmatrix}}^{B_0} + \begin{bmatrix} 47 & 31 \\ 29 & 74 \end{bmatrix}^{-1} \overbrace{\begin{bmatrix} -18 & -34 \\ -38 & -72 \end{bmatrix}}^{R_0} X.$$

Then, standard quadratic lifting gives the following equivalence.

$$\overbrace{\begin{bmatrix} 47 & 31 \\ 29 & 74 \end{bmatrix}^{-1}}^{A^{-1}} \equiv \overbrace{\begin{bmatrix} 6 & 11 \\ 49 & 93 \end{bmatrix}}^{B_0} \left(I + \overbrace{\begin{bmatrix} -18 & -34 \\ -38 & -72 \end{bmatrix}}^{R_0} X \right) \bmod X^2$$

However, the above relation is true only in the sense of a modular equivalence: equality does not hold. The straight-line formula $B_0(I + R_0X)$ for $\mathrm{Rem}(A^{-1}, X^2)$ contains the overflow term EX^2.

$$\overbrace{\begin{bmatrix} 6 & 11 \\ 49 & 93 \end{bmatrix}}^{B_0} \left(I + \overbrace{\begin{bmatrix} -18 & -34 \\ -38 & -72 \end{bmatrix}}^{R_0} X \right) = \begin{bmatrix} -52594 & -99589 \\ -441551 & -836107 \end{bmatrix}$$

$$= \overbrace{\begin{bmatrix} -2594 & 411 \\ -1551 & 3893 \end{bmatrix}}^{\mathrm{Rem}(A^{-1}, X^2)} + \overbrace{\begin{bmatrix} -5 & -10 \\ -44 & -84 \end{bmatrix}}^{E} X^2$$

Even in the presence of this overflow, applying the defining theorem for division-free

quadratic lifting (Theorem 6) nonetheless yields the following equality.

$$\overbrace{\begin{bmatrix} 47 & 31 \\ 29 & 74 \end{bmatrix}^{-1}}^{A^{-1}} = \overbrace{\begin{bmatrix} 6 & 11 \\ 49 & 93 \end{bmatrix}}^{B_0} \left(I + \overbrace{\begin{bmatrix} -18 & -34 \\ -38 & -72 \end{bmatrix}}^{R_0} X \right) + A^{-1} \overbrace{\begin{bmatrix} -18 & -34 \\ -38 & -72 \end{bmatrix}^2}^{R_0^2} X^2$$

The above equation satisfies the preconditions for applying a step of linear X-adic lifting (Theorem 3). We compute $M_0 = \text{Rem}(A^{-1}R_0^2, X)$ and $R_1 = (1/X)(R_0^2 - AM_0)$ yielding

$$A^{-1} = \underbrace{\overbrace{\begin{bmatrix} 6 & 11 \\ 49 & 93 \end{bmatrix}}^{B_0} \left(I + \overbrace{\begin{bmatrix} -18 & -34 \\ -38 & -72 \end{bmatrix}}^{R_0} X \right) + \overbrace{\begin{bmatrix} 16 & 96 \\ 44 & 8 \end{bmatrix}}^{M_0} X^2}_{B_1} + A^{-1} \overbrace{\begin{bmatrix} -5 & -17 \\ -3 & 31 \end{bmatrix}}^{R_1} X^2 X.$$

Note that although the quadratic lifting step performed above produces the overflow term $\begin{bmatrix} -5 & -10 \\ -44 & -84 \end{bmatrix} X^2$, the linear lifting step above implicitly corrects for it. The result is a straight-line formula B_1 for $\text{Rem}(A^{-1}, X^2X)$ without any overflow:

$$\begin{aligned} B_i &= \begin{bmatrix} 6 & 11 \\ 49 & 93 \end{bmatrix} \left(1 + \begin{bmatrix} -18 & -34 \\ -38 & -72 \end{bmatrix} X \right) + \begin{bmatrix} 16 & 96 \\ 44 & 8 \end{bmatrix} X^2 \\ &= \begin{bmatrix} 107406 & 860411 \\ 998449 & 243893 \end{bmatrix} \\ &= \text{Rem} \left(\begin{bmatrix} 47 & 31 \\ 29 & 74 \end{bmatrix}^{-1}, X^2X \right). \end{aligned}$$

In general, the expression B_i may have some overflow, but with a judicious choice of lifting modulus X (based on $||A||$ and n) we can ensure the overflow will be very small. Additionally, a sufficiently large choice of X guarantees that the coefficients R_i in the sparse inverse expansion are indeed of small, bounded size.

A complete statement of the double-plus-one lifting algorithm is shown as Algorithm 4 (`doublePlusOneLift`). The following theorem shows correctness and makes the claims about the size of the coefficients precise.

Algorithm 4 `doublePlusOneLift`(A, X, k)

Input: $A \in \mathbb{Z}^{n\times n}$; $X \in \mathbb{Z}$ with $X \perp \det A$ and $X \geq \max(10000, 3.61n^2||A||)$; $k \in \mathbb{Z}_{>0}$.
Output: $B_0, R_0 \ldots, R_k, M_0, \ldots, M_k \in \mathbb{Z}^{n\times n}$ as in (2.9).

1: $B_0 := \mathrm{Rem}(A^{-1}, X)$
2: $R_0 := (1/X)(I - AB_0)$
3: **for** $i = 0$ **to** $k - 1$ **do**
4: $\quad \overline{R} := R_i^2$
5: $\quad M_i := \mathrm{Rem}(B_0\overline{R}, X)$;
6: $\quad R_{i+1} := (1/X)(\overline{R} - AM_i)$
7: **end for**
8: **return** $B_0, R_0, R_1, \ldots, R_k, M_0, M_1, \ldots, M_k$

Theorem 7. *Algorithm 4 (*`doublePlusOneLift`*) is correct.*
Moreover, for all $0 \leq i \leq k$*, the output satisfies* $||R_i|| < 0.6001n||A||$ *and* $||B_i|| < 0.6X_i$.

Proof. We prove by induction on i that at the start of each loop iteration, we have

$$A^{-1} = B_i + A^{-1}R_iX_i, \tag{2.10}$$

and

$$||B_i|| < 0.6X_i, \tag{2.11}$$

where $B_i = B_{i-1}(I+R_{i-1}X_{i-1})+M_{i-1}X_{i-1}^2$. For the base case $i = 0$, Theorem 3 shows (2.10) and Lemma 1 shows (2.11). Now assume that (2.10) and (2.11) hold for some i, $i \geq 0$. Theorem 6 gives

$$A^{-1} = B_i(I + R_iX_i) + A^{-1}R_i^2X_i^2 \tag{2.12}$$

and Theorem 3 gives

$$A^{-1} = B_i(I + R_iX_i) + M_iX_i^2 + A^{-1}R_{i+1}X_{i+1},$$

which shows that (2.10) is satisfied for $i + 1$.

Equation (2.10) gives an expression and a bound for R_i:

$$\begin{aligned} R_i &= (1/X_i)(I - AB_i) \\ ||R_i|| &< (1/X_i) + 0.6n||A|| \\ &\leq 0.6001n||A|| \end{aligned}$$

This bound for $||R_i||$ gives

$$\begin{aligned} ||B_i(I + R_iX_i)|| &\leq ||B_i|| + n||B_i||||R_i||X_i \\ &< 0.6X_i + n(0.6X_i)(0.6001n||A||)X_i \\ &= (0.6/X_i + 0.36006n^2||A||)X_i^2 \\ &\leq 0.36012n^2||A||X_i^2. \end{aligned} \tag{2.13}$$

Finally, note that

$$\begin{aligned} ||B_{i+1}||/X_{i+1} &= ||B_i(I + R_iX_i) + M_iX_i^2||/(X_i^2X) \\ &\leq 0.36012n^2||A||/X + 1/2 && \text{by (2.13) and Lemma 1} \\ &< 0.6 && \text{as } X \geq 3.61n^2||A|| \end{aligned}$$

This shows that (2.11) holds for $i + 1$. □

Double-plus-one lifting performs k lifting steps, the primary cost of each being the three matrix multiplications. Theorem 7 further gives that the size of each operands in the multiplications A, B_0, R_i, M_i are bounded by $O(\log X + \log n)$. The lifting steps, then, require $O(kn^\omega \mathsf{M}(\log X + \log n))$ bit operations. It remains to account for the cost of the initial computation of $\mathrm{Rem}(A^{-1}, X)$. This quantity can be computed with $O(n^\omega(\log n)\mathsf{M}(\log X) + n^2(\log n)\mathsf{B}(\log X))$ bit operations [38, §2] by applying a unimodular triangularization algorithm [17], inverting the diagonal elements, and inverting the resulting triangular matrix by standard means [8, §28].

Combining the above gives the following corollary bounding the running time of the double-plus-one lifting algorithm.

Corollary 8. *If $k \in O(\log n)$ and $\log X \in O(\log n + \log ||A||)$ then the running time of Algorithm 4 (`doublePlusOneLift`) is bounded by $O((\log n)n^\omega\, \mathsf{M}(\log n + \log ||A||) + n^2(\log n)\mathsf{B}(\log n + \log ||A||))$ bit operations.*

The following corollary speaks to the finiteness of the sparse inverse expansion of a unimodular matrix and will be useful in the next section. As A is unimodular, A^{-1} is integral and the algorithm will compute, after $k - 1$ iterations, $B_{k-1} = A^{-1} + EX_{k-1}^2$ for some integer matrix E. If k, and, in turn, X_k, are large enough compared to $||A^{-1}||$, this error term will be small relative to X. The application of the next linear lifting step will compute the error exactly and, consequently, all subsequent coefficients in the sparse inverse expansion will be zero.

Corollary 9. *If k is chosen large enough to satisfy*

$$X_{k-1}^2 \geq (n^{(n-1)/2}||A||^{n-1})/(n^2||A||),$$

then A is unimodular if and only if R_k is the zero matrix.

Proof. First, assume $R_k = 0$. Then, by equation (2.10), $A^{-1} = B_k$ and is integral. Since $\det(A)\det(A^{-1}) = 1$ and both determinants are integral as both matrices are, $\det(A) = \pm 1$.

Next, assume A is unimodular. By Cramer's Rule and A's unimodularity, each entry in A^{-1} is an $(n-1) \times (n-1)$ minor of A. Applying Hadamard's inequality to A^{-1},

$$\begin{aligned} ||A^{-1}|| &\leq (n-1)^{(n-1)/2}||A||^{n-1} \\ &\leq (n^2||A||)X_{k-1}^2. \end{aligned} \tag{2.14}$$

Consider the application of quadratic lifting at step $k-1$ of double-plus-one lifting. At this point, $B_{k-1}(I + R_{k-1}X_{k-1})$ is congruent to A^{-1} mod X_{k-1}^2. Equivalently,

$$A^{-1} = B_{k-1}(I + R_{k-1}X_{k-1}) + EX_{k-1}^2 \tag{2.15}$$

for some error term $E \in \mathbb{Z}^{n\times n}$.

Solving (2.15) for E, yields

$$E = \left(A^{-1} + B_{k-1}(I + R_{k-1}X_{k-1})\right)/X_{k-1}^2$$

Based on the presumed size of X_{k-1}^2 and the bound on B_{k-1} and R_{k-1}, the size of the error term can be bounded as follows.

$$\begin{aligned} ||E|| &= ||A^{-1} - B_{k-1}(I + R_{k-1}X_{k-1})||/X_{k-1}^2 \\ &\leq ||A^{-1}||/X_{k-1}^2 + ||B_{k-1}(I + R_{k-1})|| \\ &\leq n^2||A|| + 0.36012n^2||A|| && \text{by (2.13) and (2.14)} \\ &< X/2 && \text{as } X \geq 3.61n^2||A|| \text{ by assumption.} \end{aligned}$$

Two previous expressions for A^{-1} relate the error E to the penultimate residue R_{k-1}:

$$\begin{aligned} A^{-1} &= B_{k-1}(I + R_{k-1}X_{k-1}) + A^{-1}R_{k-1}^2X_{k-1}^2 && \text{by (2.12)} \\ A^{-1} &= B_{k-1}(I + R_{k-1}X_{k-1}) + EX_{k-1}^2 && \text{by (2.15)} \end{aligned}$$

and therefore,

$$A^{-1}R_{k-1}^2 = E. \tag{2.16}$$

The step of linear X-adic lifting during the last iteration of the algorithm computes the linear correction M_{k-1}. Given the previous equation and the bound on the size of error term E, M_{k-1} exactly captures the error:

$$M_{k-1} = \text{Rem}(A^{-1}R_{k-1}^2, X) = \text{Rem}(E, X) = E$$

where, the last equality follows from $||E|| < X/2$ and Lemma 2.

Finally,

$$\begin{aligned} R_k &= R_{k-1}^2 - AM_{k-1} \\ &= R_{k-1}^2 - AE && \text{as } M_{k-1} = E \\ &= 0 && \text{by (2.16)} \end{aligned}$$

□

2.4 Deterministic unimodularity certification

Algorithm `uniCert`, shown in Algorithm 5, modifies `doublePlusOneLift` from the previous section to deterministically assay if a given matrix A is unimodular. As presented previously, `doublePlusOneLift` requires the lifting modulus X to be given as input. Generally and commonly, X is chosen randomly. To achieve full determinism, algorithm `uniCert` selects the modulus X to be a power of 2. Additionally, the $O(\log n)$ matrices comprising the sparse inverse expansion (R_i and M_i) are not saved during the computation: they are unnecessary for this application as only the final residue R_k is of interest. The algorithm's main loop also checks for a zero residue R_i occurring earlier than expected, as is the case when the *a priori* bound for $||A^{-1}||$ is pessimistic.

Theorem 10. *Algorithm* `UniCert` *is correct.*
The cost of the algorithm is $O((\log n)n^\omega \mathsf{M}(\log n + \log ||A||))$ *bit operations and the intermediate space requirement is bounded by* $O(n^2(\log n + \log ||A||))$ *bits.*

Proof. The running time bound follows from Corollary 8 by noting that B_0 can be computed in $O(n^\omega(\log n) + n^\omega \mathsf{M}(e))$ bit operations by first computing $\text{Rem}(A^{-1}, 2)$ and then doubling the precision up to $2^{\lceil \log_2 e \rceil}$ using algebraic Newton iteration.

Algorithm 5 `uniCert`(A)

Input: $A \in \mathbb{Z}^{n \times n}$.

Output: "Yes" if A is unimodular, "No" otherwise.

1: Let $X = 2^e$ with e minimal such that

$$X \geq \max(10000, 3.61n^2||A||).$$

2: Let $k \in \mathbb{Z}$ be minimal such that

$$X^{2^{k+1}-2} \geq (n^{(n-1)/2}||A||^{n-1})/(n^2||A||).$$

3: **if** $\det \mathrm{Rem}(A, 2) = 0$ **then return** "No"; **end if**
4: $B_0 := \mathrm{Rem}(A^{-1}, X)$
5: $R := (1/X)(I - AB_0)$
6: **for** $i = 0$ to $k - 1$ **do**
7: $\quad \overline{R} := R^2$
8: $\quad M := \mathrm{Rem}(B_0\overline{R}, X)$
9: $\quad R := (1/X)(\overline{R} - AM)$
10: $\quad$ **if** R is the zero matrix **then return** "Yes"; **end if**
11: **end for**
12: **return** "No"

We now prove correctness. Note that the algorithm computes the same quantities as algorithm `doublePlusOneLift`, so let M_i and R_{i+1} be the matrices M and R computed in iteration i of the loop. At the end of loop iteration i the algorithm has computed a sparse inverse formula B_{i+1} such that $A^{-1} = B_{i+1} + A^{-1}R_{i+1}$. If R_{i+1} is zero then A^{-1} is evidently integral; this shows that the algorithm returns "Yes" only if A is unimodular.

By definition, $X_{k-1} = X^{2^k-1}$ and hence $X_{k-1}^2 = X^{2^{k+1}-2}$. By the choice of k in the algorithm, the condition on X_{k-1}^2 stipulated by Corollary 9 holds, and if A is unimodular the algorithm will return "Yes" before completing loop iteration $i = k - 1$. □

2.5 Conclusions and future work

The algorithm presented in this chapter is an interleaving of the well-known techniques of linear and quadratic X-adic lifting. The result — "double-plus-one" lifting — is an asymptotically fast, $O(n^\omega \log n \mathsf{M}(\log n + \log ||A||))$, method for computing a sparse representation of $B = A^{-1} \bmod X^k$ and a high-order residue $R \in \mathbb{Z}^{n\times n}$ with $AB = I + RX^k$. Double-plus-one lifting reduces the computation to a small number of matrix multiplications per lifting iteration, of which there are fewer than $\log_2 n$. An implementation of the algorithm is described in Chapter 4 and demonstrates the effectiveness of the approach.

2.5.1 Comparison with shifted number system

For the sake of context, we conclude by offering a brief comparison of double-plus-one method with the previously known, asymptotically fast shifted number system method for high-order lifting [38]. Double-plus-one lifting achieves the same asymptotic complexity but offers several other advantages.

The existence of carries — small changes in the low-order digits propagating to the high-order digits — is a troublesome and inveterate property of integer arithmetic and computation. For example, suppose $X = 10$, and consider perturbing the X-adic expansion of an integer a with a small perturbation γ:

$$\overbrace{5X^4 + 9X^3 + 9X^2 + 8X + 9}^{a = 59989} + \overbrace{9X + 9}^{\gamma = 99} = \overbrace{6X^4 + 8X + 8}^{a+\gamma = 60088}.$$

Even though the perturbation is small (i.e., $\gamma \in O(X^2)$), the high-order coefficients (X^4 and X^3) of $a+\gamma$ differ from those of a. Note that because $X = 10$, this example corresponds exactly to paper-and-pencil additional of the decimal representation of a and γ.

The shifted number system is rooted in the idea that the set of representatives for the integers modulo X are ostensibly arbitrary: rather than working in the positive range $(0, 1, \ldots, p-1)$ or the symmetric range $(-\left\lfloor \frac{p-1}{2} \right\rfloor, \ldots, \left\lfloor \frac{p}{2} \right\rfloor)$ (as double-plus-one lifting does) the set of representatives are chosen to be a random set of p consecutive integers between $-p$ and p. Returning to the example, one such shifted number system uses representatives in the range $[-2, 7]$. In this system, we have

$$\overbrace{6X^4 - X - 1}^{a = 59989} + \overbrace{X^2 - 1}^{\gamma = 99} = \overbrace{6X^4 + X^2 - X - 2}^{a + \gamma}.$$

The leading coefficient of a has not been affected by the perturbation. This is true in general: the randomized shifted number system avoids and can detect the presence of error-producing carries.

Given a lifting modulus X, double-plus-one lifting is deterministic while the shifted number system method requires randomization. Additionally, the probability of failure in the shifted number system is dependent on X: a larger modulus admits more choices for the shift in representatives. As a consequence, the minimum suitable magnitude of X depends not only on the input and lifting precision required, but on the total number of computations performed in the shifted number system: more computations permit more opportunities for error to arise. Checking for carries requires the additional, explicit computation of guard coefficients. Finally, double-plus-one lifting requires a modulus size of only $O(n^2||A||)$; the shifted number system method, however, requires a modulus $X \in O(n^4 \log n||A||)$ [38, §7]. A smaller modulus means each lifting step is proportionally less expensive.

The double-plus-one lifting method is also comparatively simple. An efficient implementation is described in Chapter 4. Whereas, the complexity of the shifted number system and certain concessions made to achieve a strong theoretical result at the expense of pragmatism — embedding the input matrix in a larger matrix of power-of-two dimension, for one — render it a less desirable candidate for an efficient implementation.

2.5.2 Further remarks

Double-plus-one lifting and the high-order residue are used to give a deterministic method for unimodularity certification (testing if $\det A = \pm 1$) in section 2.4. Presumably, these techniques could also be applied to certify integrality of $A^{-1}C$ (given matrices A and C). Unimodularity and integrality certification may be further applicable to verifying the output of algorithms that would otherwise be Monte Carlo. For instance, the following chapter

relies on unimodularity certification to give a Las Vegas algorithm for the determinant and Hermite normal form.

One quibble with algorithm `doublePlusOneLift` is that it produces a sparse inverse formula that requires $\Omega(\log n)$ as much space as required to write down the input matrix. For an input matrix with dimension 1000, the space required for the sparse inverse expansion exceeds the size of the input matrix by a factor of ten. However, for applications like unimodularity and integrality certification, only the final residue R is required. Moreover, for linear system solving, the components $(R_0, M_0), (R_1, M_1), \ldots$ of the sparse inverse expansion of B could be applied as they are computed, avoiding the need to keep them.

A further refinement would be to devise a method which avoids the need to randomly find a lifting modulus X relatively prime to $\det A$. Although triangular x-basis decompositions [16] provide such a method for polynomial matrices, integers are not polynomials and no such method is known for integer matrices.

Chapter 3

Fast Heuristic Hermite Normal Form

> *Note that*'s another like compulsitory [sic] term.
>
> David Foster Wallace. *Infinite Jest* (1996).

This chapter presents a heuristic, but certified algorithm for computing the determinant and Hermite normal form of a square, nonsingular, integer matrix. The algorithm is centred on efficiently solving a series of random linear systems and a procedure for extending the affiliated solutions to a series of triangular factors of the Hermite normal form. The fast unimodularity certification and the high-order residue detailed in the previous chapter serve also key roles. Although no asymptotic time complexity is given, empirical results from an optimized implementation show the running time grows approximately as $n^3 \log n$, even for input matrices with atypical Hermite normal forms. An earlier version of this chapter's content appears in [29].

3.1 Background

Determinant

Computing the exact determinant of a nonsingular integer matrix $A \in \mathbb{Z}^{n \times n}$ is a classical problem. The determinant problem has an obvious property common to many others in computer algebra: the size of the determinant may be much larger than the size of entries in the input matrix. Considering a diagonal matrix quickly gives $\log |\det A| \geq n \log ||A||$:

the bit length of the determinant may exceed the size of entries in the input by a factor of n. However, Hadamard's inequality bounds the size of the determinant from above as $\log|\det A| \leq (n/2)\log n + n\log||A||$. Thus, a well-known modular approach based on the Chinese remainder theorem gives a deterministic algorithm for the determinant requiring $O(n^4(\log n + \log||A||^2))$ bit operations using standard, quadratic, integer arithmetic [11, §5.5].

Improving upon this first result has been the target of much study. Recent surveys [21, 22] offer a more thorough treatment, a few highlights are mentioned here. An initial significant improvement on the standard modular algorithm is a division-free algorithm to compute the determinant of a matrix over a ring [20]: the algorithm can be adapted to compute $\det A$ in $O^{\sim}(n^{3.5}\log||A||)$ bit operations; further refinements improve the exponent of n from 3.5 to 3.2 [22] assuming cubic matrix multiplication. As alluded to in the opening of the previous chapter, high-order lifting in the shifted number system gives an algorithm requiring only $O^{\sim}(n^3\log||A||)$ bit operations (again, assuming standard matrix multiplication) [38, §11]. While this matches the theoretical complexity of matrix multiplication, its practical complexity (complexity, here, in a colloquial sense) precludes a highly efficient implementation (see Section 2.5.1).

Matrix normal forms

Definition 11 (Hermite normal form)**.** *A nonsingular matrix $H \in \mathbb{Z}^{n\times n}$ is said to be in* Hermite normal form *if the following conditions are satisfied.*

- *H is upper triangular*
- *$h_{ii} > 0$, for $1 \leq i \leq n$*
- *$0 \leq m_{ij} \leq m_{ii}$ for all $1 \leq i \leq n$ and $i < j \leq n$*

That is, H is upper triangular with positive diagonal entries, and all off-diagonal entries strictly less than the diagonal entry in that column [7, Definition 2.4.2]. Although this definition is generally extended to rectangular matrices of arbitrary rank, we focus only on the square nonsingular case here.

The Hermite form is canonical: for every square nonsingular matrix A, there exists a unique matrix $H \in \mathbb{Z}^{n\times n}$ in Hermite form such that $H = UA$ for some unimodular matrix $U \in \mathbb{Z}^{n\times n}$. Any two matrices A and B related by left-multiplication by a unimodular matrix $B = UA$ are said to be left-equivalent. The Hermite form, then, is a canonical

representative of an equivalence class of left-equivalent matrices. Note also that if H is the Hermite normal form of A, the product of the first $i \leq n$ diagonal entries in H is equal to the greatest common divisor of all $i \times i$ minors in the first i columns of A [1]

The Hermite form along with the existence and uniqueness results originate with the eponymous mathematician [18]. A matrix can always be transformed to its corresponding Hermite form simply by a series of unimodular row operations [26, Theorem II.6] (elementary row operations that do not change the magnitude of the determinant). A naïve application of this approach may cause large growth in the entries of the work matrix; a polynomial time algorithm derived from it was first presented by Kannan and Bachem [23]. Hafner and McCurley give a refined algorithm requiring $O\tilde{}(n^3\mathsf{B}(n(\log n + \log ||A||))) \in O\tilde{}(n^4(\log n + \log ||A||)^2)$ bit operations [17]. Incorporating blocking and fast matrix multiplication, Storjohann and Labahn give an algorithm requiring only $O\tilde{}(n^\omega\mathsf{B}(n(\log n + log||A||))$ operations [39]. An algorithm for Hermite normal form with asymptotic complexity cubic in n (let alone n^ω) remains a desirable goal. Heuristic algorithms relying on the input matrix commonly having some pleasant property [24, 30], including the algorithm presented here, do approach this goal, at least in practice.

The analog of the Hermite form for the equivalence class of matrices under unimodular pre- *and* post-multiplication is the diagonal Smith normal form. The related concepts of determinantal divisors and, especially so, invariant factors are most relevant for our purposes.

Definition 12 (Invariant factors)**.** *For a matrix $A \in \mathbb{Z}^{n\times n}$ and $1 \leq i \leq n$, the ith* determinantal divisor α_i *is the greatest common divisor of all $i \times i$ minors of A. The ratios of successive determinantal divisors are the* invariant factors $s_i := \frac{\alpha_i}{\alpha_{i-1}}$*, with $\alpha_0 = 1$ for convenience.*

The first determinantal divisor α_1 is the greatest common divisor of the entries of A and the nth divisor α_n is the determinant $|\det A|$ itself. The diagonal matrix $S = \mathrm{diag}(s_1, s_2, \ldots, s_n)$ is the *Smith normal form* of A. Like the Hermite form, this is a canonical form. For every $A \in \mathbb{Z}^{n\times n}$, there exists a unique S in Smith form with $S = UAV$ for unimodular transforms U and V. Following directly from the definition, note that, for all i, each invariant factor s_i is integral and s_{i-1} divides s_i [26, §II.16].

[1] This seemingly esoteric fact will reappear in section 3.3 in regards to the development of a central component of this chapter's algorithms.

3.1.1 Invariant factors through linear system solving

The delightful idea that solving a system of linear equations — i.e., given $A \in \mathbb{Z}^{n\times n}$ and $v \in \mathbb{Z}^{n\times 1}$, finding a rational vector $x \in \mathbb{Q}^{n\times 1}$ satisfying $Ax = v$ — with a random right-hand side v can extract information about the invariant factors of A recurs in several contexts ([27] is an early appearance). Here, the solution x is often denoted $A^{-1}v$, though this need not require the computation of an exact inverse A^{-1}. We also call $A^{-1}v$ a projection of the inverse.

The least positive integer d such that dx is integral is called the system denominator. Equivalently, the system denominator d is the least common multiple of the denominators of the entries of x. This value in particular conveys useful information about the original matrix A. To start, we note that d always divides $\det A$. Cramer's rule gives that the ith entry of x is $x_i = \frac{\det A_i}{\det A}$, where A_i denotes A with the ith column replaced by b. Though $\det A_i$ and $\det A$ may share some common factors, the denominator of each x_i must be a factor of $\det A$: thus, the system denominator must also be. Though this fact is perhaps of little immediate use, further results in this vein refine the claim.

Abbott, Bronstein, and Mulders [1] show that the least common denominator of $A^{-1}v$ is always a divisor of the largest invariant factor s_n. Their heuristic determinant algorithm is based on this fact and the well-known phenomenon that the largest invariant factor of the matrix is often a large factor of the determinant.

For randomly chosen $v_1, v_2 \in \mathbb{Z}^{n\times 2}$, the minimal $s \in \mathbb{Z}_{>0}$ such that both $sA^{-1}v_1$ and $sA^{-1}v_2$ are integral is likely to be equal to s_n, or at least a large factor, thus decreasing the number of images of the determinant that need to be computed using the classical Chinese remainder based-determinant algorithm mentioned above. Consider the following nonsingular input matrix:

$$A = \begin{bmatrix} 33 & 8 & -50 & 45 & -38 \\ -20 & 62 & 39 & 11 & -79 \\ 13 & -82 & -52 & -65 & -37 \\ -35 & -81 & 3 & 114 & 7 \\ -100 & 14 & -114 & -22 & -10 \end{bmatrix}. \tag{3.1}$$

If we choose

$$\begin{bmatrix} v_1 & v_2 \end{bmatrix} = \begin{bmatrix} 10 & 45 \\ -16 & -81 \\ -9 & -38 \\ -50 & -18 \\ -22 & 87 \end{bmatrix}$$

then

$$A^{-1}\begin{bmatrix} v_1 & v_2 \end{bmatrix} = \begin{bmatrix} \frac{1428470455}{3313087328} & \frac{43150207}{161614016} \\ \frac{673936589}{2484815496} & \frac{66351701}{121210512} \\ -\frac{1462901509}{9939261984} & -\frac{516047293}{484842048} \\ -\frac{1221838091}{9939261984} & \frac{138504781}{484842048} \\ \frac{89642859}{414135916} & \frac{18215255}{20201752} \end{bmatrix}.$$

The denominators of $A^{-1}\begin{bmatrix} v_1 & v_2 \end{bmatrix}$ have least common multiple $s = 19878523968$, which for this example is actually equal to $-\det A$. Even if the heuristic finds a large factor of the determinant or the complete determinant itself, the running time of the method is still quartic in n, because the expected bit length of the gap between $|\det A|$ and Hadamard's inequality (and thus the bit length of the images needed by the homomorphic imaging scheme to guarantee to compute the correct determinant) is $\Theta(n)$ [1, Section 3].

Eberly, Giesbrecht, and Villard [10] make precise the claim that the system denominator of $A^{-1}v$ is likely equal to the largest invariant factor s_n. Provided $v \in \mathbb{Z}^{n\times 2}$ is chosen randomly with elements from a suitable range, $A^{-1}v$ has denominator $d = s_n$ with probability at least $1/3$. From this premise, relaxing the requirement that the determinant should be certified correct, and using additive pre-conditioners combined with binary search, they further present a Monte Carlo algorithm requiring

$$\tilde{O}(n^3(\log||A||)^2\sqrt{\log|\det A|}) \tag{3.2}$$

bit operations to recover the Smith form of A. Note that an algorithm for the Smith form also allows recovery of the determinant as $|\det A| = \prod_i^n s_i$. Using Hadamard's inequality for $|\det A|$ in (3.2) gives a worst case complexity of $\tilde{O}(n^{3.5}(\log||A||)^{2.5})$ bit operations. However, like many of these projection-based methods, the performance of the algorithm is also highly sensitive to the number of non-trivial invariant factors; assuming Hadamard's inequality gives a pessimistic estimate in comparison to the average case. For an input matrix with only $O(1)$ non-trivial invariant factors, the running time improves to $O(n^3(\log n + \log||A||)^2(\log n))$ bit operations. Importantly, further analysis by the same authors shows that an integer matrix with random entries chosen independently and uniformly from a range of λ consecutive integers is expected to have only $O(1)$ non-trivial invariant factors, provided $\lambda \in \Omega(n)$.

So far, the methods discussed have relied entirely on the denominator of the random projection. The numerators, however, can also provide further information about the invariant structure of the input matrix. One such approach, due to Wan [40, §5.6], uses the entirety of the projection to recover the largest *two* invariant factors with the same cost.

The algorithm described in the remainder of this chapter also makes use of the numerators, but is distinctly different, in goals and in methods, from the method of Wan.

3.2 Overview

The remainder of this chapter describes algorithms for the determinant and Hermite normal form that rely on, like the methods of latter half of the previous section, computing $A^{-1}v$ to extract information about the invariant structure. The algorithms presented here are sensitive to the number of non-trivial invariant factors. For propitious inputs (viz. matrices with very few non-trivial invariant factors), they perform exceptionally well. Matrices with a highly non-trivial invariant structure require relatively more work, but can still be dealt with effectively. Both algorithms certify correctness of their computation: they are randomized in the Las Vegas sense.

The premise of the algorithm is best illustrated by reprising the example in (3.1). Consider the first random projection $A^{-1}v_1$. The corresponding system denominator is $d_1 = 9939261984$. Here, d_1 is a large factor of $\det A$; in fact, $2d_1 = \det A$. Typically, only the denominators of $A^{-1}v_1$ are of interest. Instead, we also make use of the vector of numerators to produce an upper triangular basis T_1 of a super-lattice of the integer lattice generated by the rows of A. In particular, T_1 is upper triangular with minimal magnitude determinant such that $T_1A^{-1}v_1$ is integral; section 3.3 details an algorithm for computing such a matrix.

For this example, $A^{-1}v_1$ induces

$$T_1 = \begin{bmatrix} 1 & & & 15 & 183835840 \\ & 1 & & 4 & 294625615 \\ & & 1 & 1 & 159758078 \\ & & & 24 & 300295265 \\ & & & & 414135916 \end{bmatrix}.$$

The matrix T_1 has positive diagonal entries and the off-diagonal entries in each column are nonnegative and strictly less than the diagonal entries in the same column; that is, T_1 is in Hermite form. Continuing with the example, we can use the second projection $A^{-1}v_2$

to compute a second triangular factor T_2, the minimal triangular denominator of

$$T_1 A^{-1} v_2 = \begin{bmatrix} \frac{165758732}{1} \\ \frac{531308447}{2} \\ \frac{144048601}{1} \\ \frac{541532731}{2} \\ \frac{746825455}{2} \end{bmatrix}. \tag{3.3}$$

We obtain

$$T_2 = \begin{bmatrix} 1 & & & & 0 \\ & 1 & & & 1 \\ & & 1 & & 0 \\ & & & 1 & 1 \\ & & & & 2 \end{bmatrix}.$$

Since the rows of T_1 generate a super-lattice of the lattice generated by the rows of A, we can remove T_1 from A by computing

$$B_1 = AT_1^{-1} = \begin{bmatrix} 33 & 8 & -50 & -18 & 12 \\ -20 & 62 & 39 & 1 & -51 \\ 13 & -82 & -52 & 5 & 69 \\ -35 & -81 & 3 & 40 & 43 \\ -100 & 14 & -114 & 64 & 32 \end{bmatrix}.$$

The factor T_2 can also be removed to obtain

$$B_2 = B_1 T_2^{-1} = \begin{bmatrix} 33 & 8 & -50 & -18 & 11 \\ -20 & 62 & 39 & 1 & -57 \\ 13 & -82 & -52 & 5 & 73 \\ -35 & -81 & 3 & 40 & 42 \\ -100 & 14 & -114 & 64 & -23 \end{bmatrix}.$$

If, after removing the triangular factors, the remaining matrix (B_2, here) is unimodular, no further projections are necessary. In this case, $\det B_2 = \pm 1$ and so $(\det T_1)(\det T_2)$ is equal, up to sign, to the determinant of the original matrix A. (If B_2 is unimodular, the sign of the determinant can be recovered by computing $\det B_2 \bmod p$ for single odd prime p.) Likewise, $T_2 T_1$ is left-equivalent (i.e., equivalent up to pre-multiplication by a unimodular matrix) to A and hence to the Hermite normal form of A. Conveniently, the

fast unimodularity certification process of Section 2.4 can be used to determine if B_2 is unimodular.

If B_2 were not unimodular, the process continues until the work matrix is determined to be unimodular. That is, additional projections $B_2^{-1}v_3, B_3^{-1}v_4, \ldots$ can be computed and used to find corresponding triangular matrices $T_3, T_4, \ldots$ which can be factored from the work matrix as $B_3 = B_2T_3^{-1}, B_4 = B_3T_4^{-1}, \ldots$. Again, once the work matrix B_k is certified unimodular, the determinant of A is known, up to sign, with certainty.

Roughly speaking, each projection corresponds to a single invariant factor of the original matrix and the largest remaining invariant factor in the work matrix. Considering the process outlined above, each projection and corresponding triangular factor can be thought of as removing the largest remaining invariant factor. This means that the algorithm's performance is sensitive to the number of non-trivial invariant factors in the input. Random matrices often have only a single non-trivial invariant factor and, as such, often only a single projection is required.

Matrices with many non-trivial invariant factors, however, may require many projections. In fact, for a matrix with $k \in \Omega(n)$ non-trivial invariant factors, the method as sketched above requires solving the same number ($\Omega(n)$) of linear systems, equivalent in cost to computing the exact inverse of A. Obtaining an effective algorithm in this difficult case requires more care.

A main computational task used in the approach just described is to solve nonsingular linear systems to compute projections $A^{-1}v$ for some $v \in \mathbb{Z}^{n\times m}$. The most efficient algorithms for nonsingular linear system solving are based on linear p-adic lifting [6, 9, 25]. The simplest variant of linear p-adic lifting has two phases. The first phase computes a low precision inverse $C := A^{-1} \bmod p$ for a prime p with $\log p \in \Theta(\log n + \log ||A||)$. This first phase thus has cost $O(n^3(\log n + \log ||A||)^2)$ bit operations assuming standard, quadratic integer arithmetic. The second phase computes a truncated p-adic expansion

$$A^{-1}v \equiv c_0 + c_1p + c_2p^2 + \cdots + c_{\ell-1}p^{\ell-1} \bmod p^\ell,$$

each $c_i \in \mathbb{Z}^{n\times m}$ with entries reduced modulo p, from which the rational solution vector can be recovered using rational number reconstruction provided the precision ℓ is high enough. The required precision depends on the size of numerators and denominators if $A^{-1}v$. Computing one of the terms in the p-adic expansion requires a constant number of pre-multiplications with C and A of matrices of dimension $n \times m$ with entries of bit length $O(\log n + \log ||A||)$. Thus, the second phase has cost $O(\ell m \times n^2(\log n + \log ||A||)^2)$. Note that if the required precision k and the number of projections m satisfy $\ell m \in O(n)$, the overall cost of producing $A^{-1}v$ is bounded by $O(n^3(\log n + \log ||A||)^2)$ bit operations.

Instead of computing the projection $T_1 A^{-1} v_2$ by first computing $A^{-1} v_2$ and then pre-multiplying by T_1, consider computing $B_1^{-1} v_2$, equal to the vector in (3.3). Since the first factor T_1 captures a large factor of the $\det A$, the $B_1 = A T_1^{-1}$ has a much smaller determinant. In turn, the denominator of the next projection $B_1^{-1} v_2$ is also small.

However, while the denominator is small, the numerator of $B_1^{-1} v_2$ is, in general, much larger. Returning to the running example, note the dramatic size difference between the numerator and denominator of the projection.

$$B_1^{-1} v_2 = \begin{bmatrix} \frac{165758732}{1} \\ \frac{531308447}{2} \\ \frac{144048601}{1} \\ \frac{54153273}{2} \\ \frac{746825455}{2} \end{bmatrix}.$$

Consequently, the number of p-adic lifting steps (i.e., the precision ℓ) required to compute the projection is also large. Continuing in this fashion does not fully leverage the prior extraction of the largest invariant factors. Ideally, subsequent projections would require lifting only to a lower precision commensurate with the smaller size of the remaining invariant factors.

This goal can be achieved by first pre-conditioning the projection with a so-called high-order residue $R \in \mathbb{Z}^{n \times n}$, as presented in section 2.3. A high-order residue R of B_1 has the felicitous property of $B^{-1} R$ and, in turn, $B^{-1} R v$ being nearly proper. For this example, one example of such a high-order residue is

$$R = \begin{bmatrix} -15 & -15 & -15 & -15 & 1 \\ 31 & 31 & 31 & 31 & 6 \\ -47 & -47 & -47 & -47 & -4 \\ -30 & -30 & -30 & -30 & 1 \\ -104 & -104 & -104 & -104 & 55 \end{bmatrix}$$

with

$$B_1^{-1} R v_2 = \begin{bmatrix} -\frac{92}{1} \\ -\frac{97}{2} \\ -\frac{92}{1} \\ -\frac{97}{2} \\ -\frac{97}{2} \end{bmatrix}.$$

The above projection will yield the same triangular factor T_2 as the unconditioned projection $B_1^{-1}v_2$, yet the former can be computed with fewer steps of p-adic lifting. This property allows the series of projections to exploit an entry size versus dimension trade-off. As the largest remaining invariant factor decreases after each projection, the precision to compute subsequent projections decreases proportionally. Given fewer lifting steps are required, subsequent projections can use a greater number of columns while incurring no additional cost. For example, the first projection may involve only a single column. The second can then use two columns as it is expected to require at most twice the precision; the third can use four columns, and so on. In particular, the p-adic lifting precision ℓ and the number of projections m will satisfy $\ell m \in O(n)$ at each phase; as ℓ decreases, m may increase proportionally.

We do not give a rigorous cost analysis of our algorithm. In the worst case, when the Hermite form has a large number of non-trivial columns, it remains unknown how to bound the cost of combining and extracting the triangular factors corresponding to a possibly large number of projections. However, if the Hermite form has only $O(1)$ non-trivial columns, the cost of the high-order residue computation dominates the cost. In this propitious case, the entire Hermite from can be recovered with a single projection and single high-order residue computation in $O(n^3(\log n + \log ||A||)^2(\log n))$ bit operations (cf. [28] and Corollary 8) assuming standard integer arithmetic.

3.3 An algorithm for minimal triangular denominator

The procedure outlined in the previous section is contingent on a procedure to extend a rational solution vector to corresponding minimal triangular denominator. While the solution vector immediately communicates the largest invariant factor s_n, it is not clear how this information is to be extracted from the input matrix nor how the invariant factor contributes to the Hermite form. The triangular representation T of the solution vector that we will compute, however, addresses both these issues: AT^{-1} has largest invariant factor s_{n-1} and the Hermite form can be written as a product of these triangular factors.

This task is complicated by the fact that a single invariant factor may split across multiple columns in the Hermite form. Moreover, a single column in the Hermite form may be composed of factors of several invariant factors. The example below illustrates the first of these occurrences. Matrix A has a single non-trivial invariant factor $s_n = 1155$, but

its Hermite form has four non-trivial diagonal elements.

$$A = \begin{bmatrix} 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & -1 & -1 & 4 \\ 0 & 3 & -3 & -1 & 1 \\ 2 & 1 & 2 & -4 & 1 \\ 5 & 1 & 3 & 2 & -1 \end{bmatrix} \quad H = \begin{bmatrix} 1 & 2 & 3 & 8 & 0 \\ & 3 & 4 & 9 & 1 \\ & & 7 & 10 & 0 \\ & & & 11 & 3 \\ & & & & 5 \end{bmatrix} \quad S = \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ & & & & 1155 \end{bmatrix}$$

This example also motivates an approach to find the triangular matrix T corresponding to a projection v. Given a rational vector $v \in \mathbb{Q}^{n\times 1}$, we find a nonsingular upper triangular matrix $T \in \mathbb{Z}^{n\times n}$ such that Tv is integral and T is of minimal magnitude determinant. Such a matrix T is said to be a *minimal triangular denominator* of v.

Matrix T clears the denominator of rational vector v and thus, in the context at hand, is closely related to the invariant factor captured by projection v. The constraint that T have minimal determinant invalidates an obvious solution $T = \text{diag}(s_n, \ldots, s_n)$ and forces the invariant factor to be split across multiple columns of T.

The following subsection outlines an algorithm for computing a minimal triangular denominator from a single solution vector $v \in \mathbb{Q}^{n\times 1}$. Then, this procedure is extended to operate on a block of rational solution vectors $v \in \mathbb{Q}^{n\times k}$.

3.3.1 Single-column minimal triangular denominator

Our algorithm to compute minimal triangular denominators is based on the following lemma relating the minimal triangular denominator to the Hermite form of a simple, specially constructed matrix.

Lemma 13. *Let $v \in \mathbb{Q}^{n\times 1}$ and $d \in \mathbb{Z}_{>0}$ be such that $w := dv$ is integral. If the Hermite form of*

$$B := \left[\begin{array}{c|c} d & \\ \hline w & I_n \end{array}\right] \in \mathbb{Z}^{(n+1)\times(n+1)} \tag{3.4}$$

is

$$\left[\begin{array}{c|c} * & * \\ \hline & H \end{array}\right] \in \mathbb{Z}^{(n+1)\times(n+1)}, \tag{3.5}$$

then $H \in \mathbb{Z}^{n\times n}$ is a minimal triangular denominator of v.

Proof. The unique unimodular matrix U that transforms B to Hermite form is given by

$$\begin{aligned} U &= \left[\begin{array}{c|c} k & * \\ \hline & H \end{array}\right] \left[\begin{array}{c|c} d & \\ \hline w & I_n \end{array}\right]^{-1} \\ &= \left[\begin{array}{c|c} *d^{-1} & * \\ \hline -Hv & H \end{array}\right] \in \mathbb{Z}^{(n+1)\times(n+1)}. \end{aligned} \tag{3.6}$$

By definition of the Hermite form, U is integral. Therefore, submatrix $-Hv$ of U shown in (3.6) is integral as required.

Next, suppose that T is a minimal triangular denominator of v: this implies $|\det T| \leq |\det H|$. Define V as U with T replacing H; that is,

$$V = \left[\begin{array}{c|c} *d^{-1} & * \\ \hline -Tv & T \end{array}\right] \in \mathbb{Z}^{(n+1)\times(n+1)}.$$

By the analogous construction of U and V, we have both $|\det U||\det B| = |k||\det H|$ and $|\det V||\det B| = |k||\det T|$. And, since $\det U = \pm 1$, $|\det V||\det H| = |\det T|$. Matrix V is integral as Tv is, thus, $|\det H| \leq |\det T|$.

Combining the above with the assumption of $|\det T|$'s minimality, it follows that $|\det H| = |\det T|$.

□

Let $v \in \mathbb{Q}^{n\times 1}$ and $d \in \mathbb{Z}_{>0}$ be such that $w := dv$ is integral, as in Lemma 13. Let us define the entries of our input vector w and target H as

$$w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} \quad \text{and} \quad H = \begin{bmatrix} h_1 & h_{12} & \cdots & h_{1n} \\ & h_2 & \cdots & h_{2n} \\ & & \ddots & \vdots \\ & & & h_n \end{bmatrix}. \tag{3.7}$$

The obvious approach to compute a minimal triangular denominator of v is to simply apply unimodular row operations to triangularize matrix

$$B = \left[\begin{array}{c|c} d & \\ \hline w & I_n \end{array}\right] \in \mathbb{Z}^{(n+1)\times(n+1)}$$

as in Lemma 13 (3.4).

The extended Euclidean algorithm induces such a unimodular row operation. Define Gcdex to be the operation that takes as input $a, b \in \mathbb{Z}$ and returns as output s, t, v, h, g such that

$$\begin{bmatrix} s & t \\ v & h \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} g \\ 0 \end{bmatrix}$$

with $sh - tv = \pm 1$ and g a greatest common divisor of a and b. We further specify that $h > 0$ and $0 \leq t < h$. Note that s and t are the Bézout coefficients as computed by the extended Euclidean algorithm and we may take $v = -|b|/g$ and $h = |a|/g$.

For instance, if we compute $s_n, t_n, v_n, h_n, g_n = \text{Gcdex}(d, w_n)$, then the following unimodular transformation of the input matrix zeroes out the entry occupied by w_n.

$$\left[\begin{array}{c|ccc} s_n & & & t_n \\ \hline & 1 & & \\ & & \ddots & \\ & & & 1 \\ v_n & & & h_n \end{array}\right] \underbrace{\left[\begin{array}{c|ccc} d & & & \\ \hline w_1 & 1 & & \\ \vdots & & \ddots & \\ w_{n-1} & & & 1 \\ w_n & & & 1 \end{array}\right]}_{B} = \left[\begin{array}{c|ccc} g_n & & & t_n \\ \hline w_1 & 1 & & \\ \vdots & & \ddots & \\ w_{n-1} & & & 1 \\ & & & h_n \end{array}\right]$$

This process can be repeated to clear the remaining off-diagonal entries $(w_{n-1}, \ldots, w_2, w_1)$ in the first column of B. Provided $B \in \mathbb{Z}^{(n+1)\times(n+1)}$ is initialized as the input matrix in (3.4), the following loop performs such a series of transformations. After the loop completes the work matrix B will be upper triangular.

$g_{n+1} := d$;
for $i = n$ **downto** 1 **do**
  $s_i, t_i, v_i, h_i, g_i := \text{Gcdex}(g_{i+1}, w_i)$;
  $\begin{bmatrix} B[1,*] \\ B[i+1,*] \end{bmatrix} := \begin{bmatrix} s_i & t_i \\ v_i & h_i \end{bmatrix} \begin{bmatrix} B[1,*] \\ B[i+1,*] \end{bmatrix}$
od

Note that this loop updates the top row of the work matrix at every iteration. Consequently, the entries in this row become too large and subsequent application of the unimodular transformations become prohibitively expensive. Indeed, at the start of iteration

$i-1$ the work matrix has the shape

$$B = \left[\begin{array}{c|ccc|ccc} g_i & & & & t_i & \cdots & * \\ \hline w_1 & 1 & & & & & \\ \vdots & & \ddots & & & & \\ w_{i-1} & & & 1 & & & \\ \hline & & & & h_i & \cdots & * \\ & & & & & \ddots & \vdots \\ & & & & & & h_n \end{array}\right] \tag{3.8}$$

with each entry in the top row potentially very large.

Algorithm 6 `hcol`(w,d)

Input: A vector $w \in \mathbb{Z}^{n\times 1}$ and $d \in \mathbb{Z}$.
Output: $H \in \mathbb{Z}^{n\times n}$, a minimal triangular denominator of w/d.

[1. Diagonal entries]
$g_{n+1} := d$;
for $i = n$ **downto** 1 **do**
  $*, t_i, *, h_i, g_i := \mathrm{Gcdex}(g_{i+1}, w_i)$
end for

[2. Off-diagonal entries]
for $i = 1$ **to** n **do**
  if $h_i = 1$ **then next fi**
  for $k = 1$ **to** $i-1$ **do**
    $h_{k,i} := \mathrm{Rem}(-t_i w_k, h_i)$
    $w_k := \mathrm{Rem}(w_k + h_{k,i} w_i, d)$
  end for
  $d := d/h_i$
  $w_j := \begin{cases} w_j/h_i & i \neq j \\ w_j & i = j \end{cases}$
end for
return H as in (3.7)

Instead, we proceed in two distinct phases. First, we use the Gcdex operation to determine only the diagonal entries h_i in decreasing order; the induced unimodular row

operations are not applied to the work matrix. Next, we compute the off-diagonal entries column-by-column, left-to-right, by appealing to the definition of H as the minimal triangular denominator of w/d.

At the start of iteration i of the second phase, the first $i-1$ columns of H have been computed. The work matrix looks as follows.

$$\begin{bmatrix} d \\ \hline \overline{w}_1 \\ \vdots \\ \overline{w}_{i-1} \\ \hline w_i \\ \vdots \\ w_n \end{bmatrix} := \left[\begin{array}{c|ccc|c} 1 & & & & \\ \hline & h_1 & \cdots & h_{1,i-1} & \\ & & \ddots & \vdots & \\ & & & h_i & \\ \hline & & & & \begin{matrix} 1 & & \\ & \ddots & \\ & & 1 \end{matrix} \end{array}\right] \begin{bmatrix} d \\ \hline w_1 \\ \vdots \\ w_{i-1} \\ \hline w_i \\ \vdots \\ w_n \end{bmatrix}$$

The algorithm maintains the invariant that, at the start of iteration i, vector $\overline{w} = Hw$ (as above) has all entries divisible by $d/(h_i h_{i+1} \dots h_n)$. During iteration i, then, the off-diagonal entries $h_{1,i}, h_{2,i}, \dots h_{i-1,i}$ are computed so as to maintain this invariant. As a result, after completion of the algorithm $\overline{w} = Hw$ has all entries divisible by d; that is, $H(w/d)$ is integral.

Algorithm `hcol`, shown in Algorithm 6, implements the above method to compute H. The following theorem demonstrates its correctness.

Theorem 14. *Algorithm* `hcol` *computes the minimal triangular denominator (in Hermite form) of* w/d.

Proof. Define $H^{(i-1)}$ and $w^{(i-1)}$ to be H and w at the start of iteration i. Define $\overline{w}^{(i)} := H^{(i)}w$. By induction on i we show

- $g_i \mid \overline{w}^{(i-1)}$
- $w^{(i-1)} \equiv \overline{w}^{(i-1)}/g_i \bmod d/g_i$.

First, as $H^{(0)} := I$, $\overline{w}^{(0)} = w$. All entries of $\overline{w}^{(0)} = w$ are divisible by g_1 since $g_1 = \gcd(d, w_n, \dots, w_2, w_1)$.

At the start of iteration i, we assume the following holds.

$$\underbrace{\left[\begin{array}{c} d \\ \hline \overline{w}_1 \\ \vdots \\ \overline{w}_{i-1} \\ \hline w_i \\ w_{i+1} \\ \vdots \\ w_n \end{array}\right]}_{\overline{w}^{(i-1)}} = \underbrace{\left[\begin{array}{ccc|c|c} 1 & & & & \\ \hline h_1 & \cdots & h_{1,i-1} & & \\ & \ddots & \vdots & & \\ & & h_{i-1} & & \\ \hline & & & 1 & \\ & & & & 1 \\ & & & & & \ddots \\ & & & & & & 1 \end{array}\right]}_{H^{(i-1)}} \underbrace{\left[\begin{array}{c} d \\ \hline w_1 \\ \vdots \\ w_{i-1} \\ \hline w_i \\ w_{i+1} \\ \vdots \\ w_n \end{array}\right]}_{w}$$

We show that, in iteration i, entries of the i column of $H^{(i)}$ are selected such that every entry of $\overline{w}^{(i)} = H^{(i)}w$ is divisible by g_{i+1}.

$$\underbrace{\left[\begin{array}{c} d \\ \hline \overline{w_1}^{(i)} \\ \vdots \\ \overline{w_{i-1}}^{(i)} \\ \hline \overline{w_i}^{(i)} \\ \hline w_{i+1} \\ \vdots \\ w_n \end{array}\right]}_{\overline{w}^{(i)}} = \underbrace{\left[\begin{array}{ccc|c|c|c} 1 & & & & & \\ \hline h_1 & \cdots & h_{1,i-1} & h_{1,i} & & \\ & \ddots & \vdots & \vdots & & \\ & & h_{i-1} & h_{i-1,i} & & \\ \hline & & & h_i & & \\ \hline & & & & 1 & \\ & & & & & \ddots \\ & & & & & & 1 \end{array}\right]}_{H^{(i)}} \underbrace{\left[\begin{array}{c} d \\ \hline w_1 \\ \vdots \\ w_{i-1} \\ \hline w_i \\ \hline w_{i+1} \\ \vdots \\ w_n \end{array}\right]}_{w}$$

The relationships below follow from the precomputations of phase 1 and will be useful in the remainder of the proof:

$$\begin{aligned} g_n &:= d \\ g_i &:= \gcd(g_{i+1}, w_i) \\ &= \gcd(w_i, w_{i+1}, \ldots, w_n, d) && (3.9) \\ g_i &= s_i g_{i+1} + t_i w_i && (3.10) \\ 0 &= v_i g_{i+1} + h_i w_i && (3.11) \end{aligned}$$

As the diagram above suggests, we show $g_{i+1} \mid \overline{w_j}^{(i)}$ by considering three cases, dependent on j.

- $j > i$
  These entries are unchanged from the original input: $\overline{w_j}^{(i)} = w_j$. By construction in (3.9), g_{i+1} is a divisor of each $w_{j>i}$.

- $j = i$
  Since h_i satisfies $0 = v_i g_{i+1} + h_i w_i$ (3.11) for some integral v_i,

$$\begin{aligned}\overline{w_j}^{(i)} &= h_i w_i \\ &= -v_i g_{i+1}.\end{aligned}$$

- $j < i$
  The off-diagonal elements in column i of H are defined as

$$h_{j,i} := \mathrm{Rem}(-t_i w_j^{(i-1)}, h_i) = -t_i w_j^{(i-1)} + q h_i,$$

  for some $q \in \mathbb{Z}$. Also, (and by assumption), $w_j^{(i-1)} := \overline{w_j}^{(i-1)}/g_i$.

$$\begin{aligned}
\overline{w_j}^{(i)} &= \sum_{k=j}^{i} w_k h_{j,k} \\
&= \overline{w_j}^{(i-1)} + h_{j,i} w_i \\
&= \overline{w_j}^{(i-1)} - t_i w_i {w_j}^{(i-1)} + q h_i w_i && \text{definition of } h_{j,i} \\
&\equiv \overline{w_j}^{(i-1)} - t_i w_i {w_j}^{(i-1)} \bmod g_{i+1} && 0 = v_i g_{i+1} + h_i w_i \\
&\equiv \overline{w_j}^{(i-1)} - g_i {w_j}^{(i-1)} && g_i = s_i g_{i+1} + t_i w_i \\
&\equiv \overline{w_j}^{(i-1)} - g_i \left(\frac{\overline{w_j}^{(i-1)}}{g_i} \right) && \text{definition of } {w_j}^{(i-1)} \\
&\equiv 0 \bmod g_{i+1}
\end{aligned}$$

In all of the above cases, $g_{i+1} \mid \overline{w_j}^{(i)}$ as required.

It remains to show that $w^{(i)} \equiv \overline{w}^{(i)}/g_{i+1} \bmod d/g_{i+1}$.

Note that d, h_i and g_i are related as follows

$$\begin{aligned}d &= h_1 h_2 \ldots h_n \text{ and} \\ g_i &= h_1 h_2 \ldots h_{i-1};\end{aligned}$$

thus,

$$\begin{aligned}
w_j^{(i)} &:= \frac{1}{h_i}\mathrm{Rem}\left(w_j^{(i-1)} + h_{j,i}w_i^{(i-1)}, d/g_i\right) \\
&= \frac{1}{h_1}\mathrm{Rem}\left(\frac{\overline{w_j}^{(i-1)} + h_{j,i}w_i}{g_i}, d/g_i\right) && \text{assumption on } w_j^{(i-1)} \\
&= \frac{1}{h_i}\mathrm{Rem}\left(\frac{\overline{w_j}^{(i)}}{g_i}, d/g_i\right) && \text{definition of } \overline{w_j}(i) \\
&= \frac{\overline{w_j}^{(i)} + qd}{h_i g_i} && \text{for some } q \in \mathbb{Z} \\
&\equiv \overline{w_j}^{(i)}/g_{i+1} \bmod d/g_{i+1} && \text{since } g_{i+1} = h_i g_i.
\end{aligned}$$

After n iterations, $\overline{w^{(n)}} = H^{(n)}w$ is divisible by $g_{n+1} = d$. Thus, $T(w/d)$ is integral as required. Moreover, $H^{(n)}$ has the same determinant as the trailing $n \times n$ block in the Hermite form of B. By Lemma 13 then, $H^{(n)}$ is a minimal triangular denominator of w/d.

□

We analyze the cost of Algorithm 6 under the assumption of standard integer arithmetic and the so-called naïve cost model [2, Chapter 3]. The number of bits in the binary representation of an integer a is given by

$$\lg a = \begin{cases} 1 & a = 0 \\ 1 + \lfloor \log_2 |a| \rfloor & a \neq 0 \end{cases}$$

For integers a and b, computing the product ab and $\mathrm{Gcdex}(a, b)$ both require $O((\lg a)(\lg b))$ bit operations in this setting. For an integer a and nonzero integer b, the operation of division with remainder (find q and r such that $a = qb + r$ and $0 \leq r < |b|$) requires $O((\lg q)(\lg b))$ bit operations. Note that the cost of division with remainder is dependent on the size of the quotient q and not on the dividend a.

Corollary 15. *If entries in $w \in \mathbb{Z}^{n\times 1}$ are reduced modulo d, then the running time of Algorithm 6 is bounded by $O(n(\log d)^2)$ bit operations.*

Proof. Let D denote the initial value of d as passed into the algorithm. Assume without loss of generality that $D > 1$.

Phase 1 performs n extended gcd computations with operands (an vector entry w_i and a divisor thereof) bounded in magnitude by D. This has cost bounded by $O(n(\log D)^2)$ bit operations.

Now consider phase 2. At the start of each iteration of the outer loop (line 5), all entries in w are reduced modulo d, a divisor of D. For the first iteration, this follows from the theorem's precondition; for later assumptions, it is enforced by the reduction on line 12. We make use of two additional facts. First, $|t_i| < h_i$ follows from the definition of Gcdex. Second, $|h_{k,i}| < h_i$ by construction (on line 8). Thus, during iteration i of the outer loop each individual arithmetic operation — one of $\{+, -, \times, /, \text{Rem}\}$ — performed in the *inner* loop uses $O((\lg D)(\lg h_i))$ bit operations. In particular, the Rem operation on line 8 requires $O((\lg D)(\lg h_i))$ bit operations as $\text{Quo}(t_i w_k, h_i) \leq w_k \leq D$. Likewise, the Rem operation on line 9 has the same cost since $\text{Quo}(w_k + h_{k,i} w_i, d) \leq h_{k,i} + 1 \leq h_i$. As the number of iterations of the inner loop in bounded by $O(n)$, there exists an absolute constant c such that iteration i of the outer loop has cost bounded by $cn(\lg D)(\lg h_i)$ bit operations.

Let $L = \{i \mid h_i > 1\}$, the set of indices for which the outer loop does work. Then the total cost of phase 2 is bounded by

$$\begin{aligned}
\sum_{i \in L} cn(\lg D)(\lg h_i) &\leq cn(1 + \log_2 D) \sum_{i \in L} (1 + \log_2 h_i) \\
&\leq cn(2 \log_2 D)(2 \sum_{i \in L} \log_2 h_i) \\
&= 4cn(\log_2 D)(\log_2 h_1 h_2 \cdots h_n) \\
&\leq 4cn(\log_2 D)(\log_2 D).
\end{aligned}$$

That is, phase 2, and thus the entire algorithm, requires at most $O(n(\log D)^2)$ bit operations. □

We remark that our determinant algorithm will call `hcol` with a sequence of vectors that have denominator $d_1, \ldots, d_k$ with $d_1 \cdots d_k = |\det A|$. The total cost of all calls to `hcol` will thus be bounded by $O(n(\log |\det A|)^2)$ bit operations, which becomes $O(n^3(\log n + \log ||A||)^2)$ in the worst case using Hadamard's inequality for $|\det A|$.

3.3.2 Block minimal triangular denominator

The method described above operates on only a single rational vector at a time. However, as alluded to in the overview, projections of multiple columns are required to best leverage the

decreasing size of the largest invariant factor. This section extends the minimal triangular denominator for a single column to one for multiple columns.

Algorithm 7 `hermiteOfProjection`(X,d)

Input: $X \in \mathbb{Z}^{n\times k}$ and $d \in \mathbb{Z}$.
Output: $H \in \mathbb{Z}^{n\times n}$, a minimal triangular denominator of X/d.

1: $H := I_n$
2: **for** $i = 1$ **to** k **do**
3: $\quad y := H(X[*, i]) \bmod d$
4: $\quad g := \mathrm{Gcd}(d, y_1, y_2, \ldots, y_n);\ d' := d/g;\ y := y/g$
5: $\quad$ **if** $d' = 1$ **then next fi**
6: $\quad T_i := \texttt{hcol}(y, d')$
7: $\quad H := T_i H$
8: **end for**
9: **return** `triangularHNF`(H)

Suppose a projection $X/d := A^{-1}V$ for $V, X \in \mathbb{Z}^{n\times k}$, V random, and $d \in \mathbb{Z}$ is given. Denote the ith column of X by X_i. Naturally, a minimal triangular denominator T_1 corresponding to the first column X_1 can be found by the method of the previous section. Extending T_1 to a minimal triangular denominator for the first two columns of X/d requires additional work.

Working with $X_2 = A^{-1}V_2$ is insufficient: $T_2 := \texttt{hcol}(X_2)$ clears the denominator of X_2/d, but not necessarily the denominator of X_1/d. The product T_1T_2 violates the requirement of a minimal determinant. Instead, we apply T_1 to X_2 before computing T_2. The result is that $T_2(T_1X_2/d)$ is integral. Moreover, since T_1X_1/d is integral, $T_2T_1\,[\ X_1\ X_2\ ]\,/d$ is as well. We extend this idea to an arbitrary number of columns by keeping a running product $T_{i-1}\ldots T_2T_1$ and applying it to the subsequent column X_i before computing a single-column minimal triangular denominator. Algorithm 7 gives the complete algorithm; `triangularHNF` refers to an algorithm for the Hermite form of a triangular matrix [36].

Finally, line 4 in Algorithm 7 deserves an additional word of explanation. After applying H to X_i, it may be that the entries of the resulting vector share some common divisor. Algorithm `hcol` requires that the greatest common divisor of each entry in the column vector and the denominator be one, line 4 simply removes any common factor from both the vector y and the denominator d.

3.4 The projection method for determinant

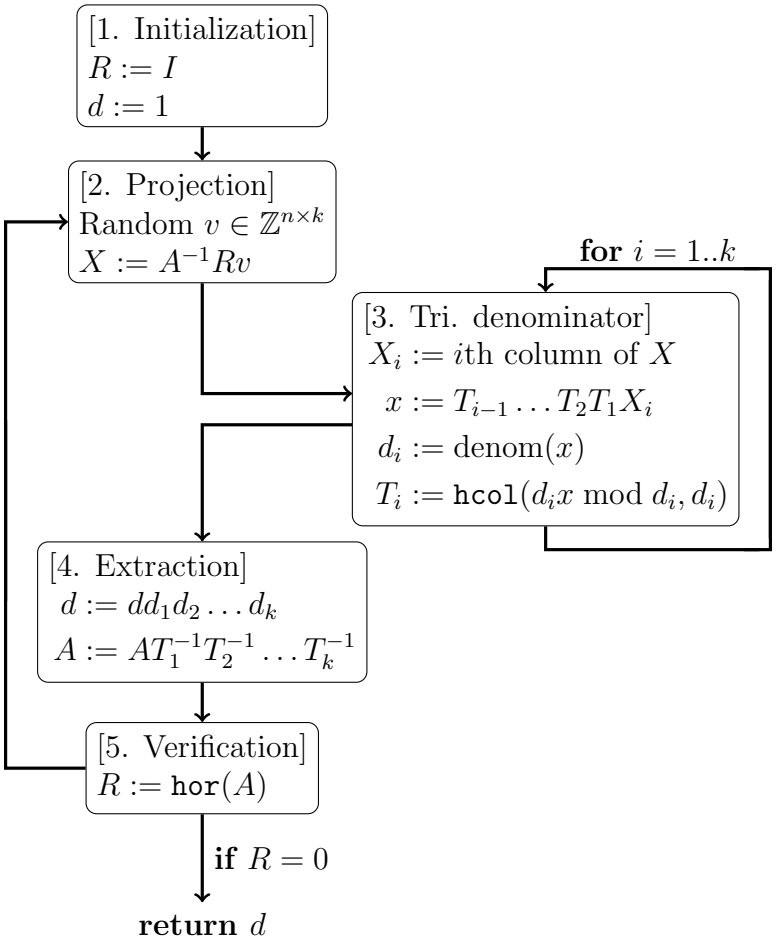

Figure 3.1: Overview of determinant algorithm.

Figure 3.1 gives a high-level overview of our determinant algorithm. The first phase uses a highly-optimized implementation of p-adic linear system solving [6] to compute $X = A^{-1}v_1$, a projection of the inverse A^{-1} for a random block of vectors $v_1 \in \mathbb{Z}^{n \times k}$. The procedure described in the previous section (Algorithm 7, `hermiteOfProjection`) computes minimal triangular denominator T_1 for X. The determinant of T_1 – equivalently, the system denominator of $A^{-1}v_1$ — is denoted d_1 and is a divisor of $\det A$. We can then proceed by considering projections of $A' = AT_1^{-1}$. The determinant of A' is equal to

$\det A/d_1$ and as the rows of T_1 generate a super-lattice of lattice generated by the rows of A, AT_1^{-1} is integral.

We may check the completeness of the determinant extracted thus far by certifying the unimodularity of A'. Applying the method of the previous chapter, the high-order residue R corresponding to A' can be computed by the method of double-plus-one lifting. If R is the zero matrix, $\det A' = \pm 1$ and, thus, $|\det A| = d_1$. Otherwise, the process continues with A' and a new random block of vectors v_2. As demonstrated in the overview section, the high-order residue R also serves to compress (in a sense) the subsequent projections: that is, we compute $A'Rv_2$ on the second iteration and do likewise until completion. Once the work matrix is certified to be unimodular, the determinant of A can be recovered as the product of the minimal triangular denominators $T_1, T_2, \ldots$.

We note that the above algorithm is randomized in the Las Vegas sense: the output is always correct, but the number of iterations required to produce that output may vary. In the case of generic matrices, the computation of the high order residue (the "Verification" phase in Figure 3.1) serves only to verify the correctness of the result. Omitting this verification step, then, yields a faster Monte Carlo randomized algorithm: one that computes the determinant from a single projection with high probability.

In theory, AT^{-1} may have entries larger than A by a factor n; though this is phenomenon does not manifest itself in practice, it precludes a suitable result for the asymptotic time complexity for this algorithm. The following lemma, however, gives a rough bound for the total number of projections required to extract the entirety of the determinant.

Lemma 16. *If $A \in \mathbb{Z}^{n\times n}$ has k non-trivial invariant factors, the number of projections ℓ required to capture all of them with probability less than $e^{-\epsilon}$ is $\ell \geq 3k + \epsilon + \sqrt{\epsilon^2 + 6k\epsilon}$*

Proof. By [10], the entirety of largest invariant factor is successfully extracted with probability $p \geq \frac{1}{3}$.

If this process is repeated ℓ times, the number of invariant factors extracted follows a binomial distribution with success probability $p = \frac{1}{3}$ and ℓ trials: $X \sim B(\ell, p)$. The probability that this process fails — that fewer than k invariant factors are extracted — is $Pr(X < k)$. It is natural to require that the process' mean ℓp exceeds k. Enforcing this constraint allows the application of the following Chernoff bound [8]:

$$Pr[X < (1-\delta)\ell p] < \exp\left(\frac{-\ell\delta^2}{2}\right)$$

with $0 < \delta < 1$. In the context here, $\delta = 1 - \frac{k}{\ell p}$, and rewriting the bound gives

$$Pr[X < k] < \exp\left(-\frac{(k - \ell p)^2}{2\ell p^2}\right).$$

For convenience, define $r = \ell p - k$, the amount by which the mean exceeds the target value. The resulting inequality is then

$$\begin{aligned} \frac{r^2}{2p(r+k)} &< \epsilon \\ 0 &< -r^2 + 2rp\epsilon + 2kp\epsilon \end{aligned}$$

To ensure a failure probability of less than $e^{-\epsilon}$,

$$r > p\epsilon + \sqrt{(p\epsilon)^2 + 2kp\epsilon}.$$

Equivalently,

$$\ell \geq \left(k + p\epsilon + \sqrt{(p\epsilon)^2 + 2kp\epsilon}\right)/p$$

Substituting $p = 1/3$ yields the desired result. □

3.5 Extension to Hermite form

A conceptually straightforward extension of the projection method allows the recovery of the entire Hermite form. Figure 3.2 gives an overview. The Hermite form algorithm differs from the preceding determinant algorithm only in the "extraction" phase.

In the determinant algorithm, the triangular denominators are repeatedly extracted from the same matrix in place (i.e., $A := AT_1^{-1} \ldots T_k^{-1}$) and then discarded. Here, each set of T_i is combined into a single work matrix H (i.e., $H := T_k \ldots T_1 H$) which eventually contains the Hermite form of A. This process may cause growth in the off-diagonal entries of H; that is, $H = T_k \ldots T_1$ may have the appropriate shape and diagonal entries, but may not be in Hermite form. A special Hermite normal form algorithm [36] (denoted `hermite` in Figure 3.2) for triangular matrices provides an efficient scheme for appropriately reducing the off-diagonal entries.

Additionally, H must be extracted from the original input matrix — not a work matrix — at each stage. That is, A is not updated in-place; rather, the subsequent projection and verification phases operate on $B := AH^{-1}$.

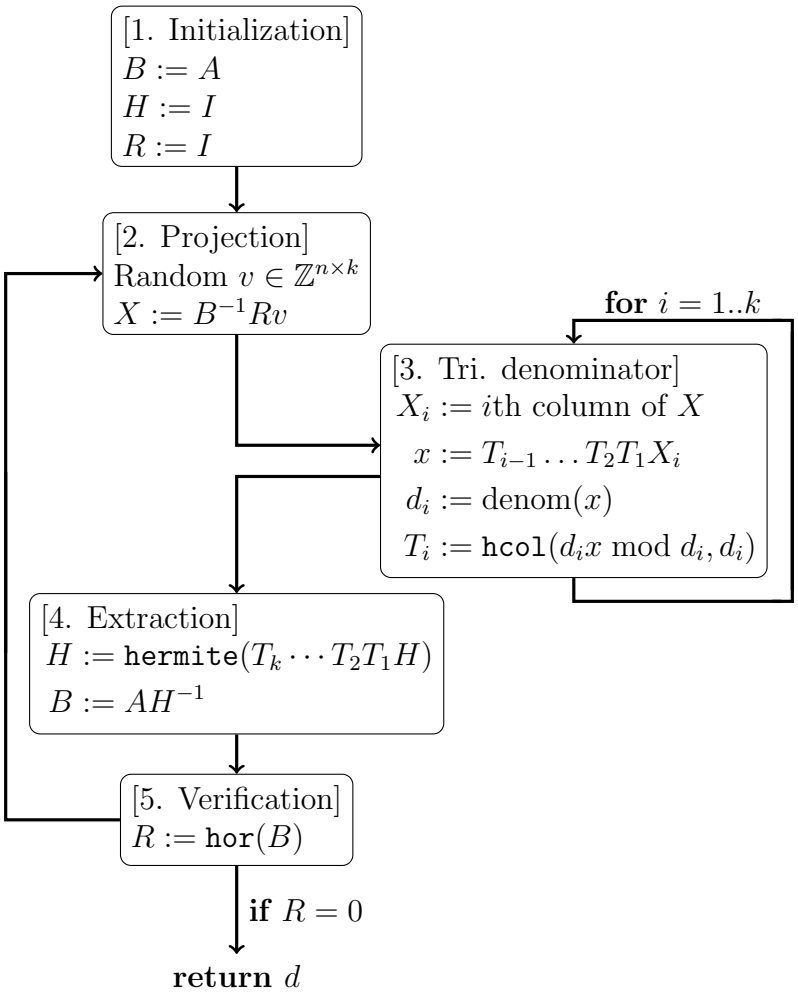

Figure 3.2: Overview of Hermite form algorithm.

The additional cost required to combine the minimal triangular denominators and extract them from the input matrix complicates attempts to obtain a strong result for the asymptotic complexity of the algorithm. Even if there are very few invariant factors, the Hermite form may have many non-trivial columns in the worst case. Secondly, preliminary analysis suggests that in the worst case the entries in AH^{-1} can have n more bits compared to the entries in A. However, this growth has not been observed in practice, even for matrices with highly non-trivial Hermite forms.

3.6 Conclusions and future work

This chapter gives heuristic algorithms for the determinant and Hermite normal form of a nonsingular integer matrix; both algorithms certified the correctness of their result and are most efficient on random input matrices. The following chapter outlines empirical results demonstrating practical effectiveness.

The central component of these algorithms is a routine for efficiently extending a projection $A^{-1}v$ to a minimal triangular denominator T such that $TA^{-1}v$ is integral. Combining the triangular factors corresponding to multiple projection yields an expression for the Hermite form up to multiplication by a unimodular matrix.

The algorithm for Hermite normal form given here can be extended to one for Smith normal form. An efficient algorithm for finding the Smith normal form of a triangular input matrix is given in [36]; this algorithm can be directly applied to the result of our Hermite form computation.

Extending our Hermite form algorithm to one efficiently handling matrices of arbitrary shape and rank requires additional care. While the tools to develop such an algorithm exist, doing so efficiently is non-trivial and may require additional techniques.

Chapter 4

Implementation and Empirical Results

What's so impressive about a diamond except the mining?

Fiona Apple. "Red Red Red", *Extraordinary Machine* (2005).

An optimized, high-performance implementation of the algorithms of the previous section augments the theoretical complexity results for double-plus-one lifting and justifies the claims of practical effectiveness for our Hermite normal form algorithm. This chapter outlines the overarching approach to our implementation, discusses several implementation concerns and optimizations of practical importance, and, finally, compares our Hermite normal form algorithm against the best-known implementations available in established computer algebra systems.

4.1 Reduction to Basic Linear Algebra Subprograms

Just as reduction to matrix multiplication is an attractive theoretical goal (i.e., demonstrate an $O^{\sim}(n^{\omega})$ algorithm for a given problem) a practically effective implementation in terms of existing matrix multiplications is similarly attractive. There exist high-performance, exceedingly optimized matrix multiplication routines, in some cases tailored to take advantage of the particulars of a specific architecture. The algorithms of the previous chapters make extensive use of either unadorned matrix multiplication or of routines that themselves build upon matrix multiplication (viz. nonsingular system solving).

The Basic Linear Algebra Subprograms (BLAS) [4] are a ubiquitous interface providing, as the name suggests, low-level linear algebra routines: scalar/vector, matrix/vector, and matrix/matrix multiplication, referred to as the level one, two, and three BLAS routines, respectively. Strictly speaking, the BLAS refer only to the interface to and a reference, non-optimized implementation of the routines themselves. The development of fast implementations is left to hardware vendors (e.g., Intel's Math Kernel Library; AMD's Core Math Library) or other motivated parties (e.g., R.C. Whaley et al.'s ATLAS BLAS [41]; K. Goto's GotoBLAS [14]).

Our implementation relies primarily on the level 3 BLAS routines provided by the Automatically Tuned Linear Algebra Software (ATLAS) library [41]. ATLAS BLAS is a widely used, portable, highly optimized implementation of the BLAS. It has the distinguishing property of being, appropriately, automatically tuned for the architecture and system on which it is compiled.

4.1.1 Residue number systems

ATLAS operates on matrices of double-precision floating-point numbers. To allow our implementation to work with matrices of arbitrarily large integers despite this limitation, we employ a standard Chinese Remainder Theorem-inspired modular scheme: operations on matrices over the integers are instead performed element-wise on their residues modulo multiple suitably-sized primes. Given a basis of pairwise coprime integers $\mathcal{P} = (p_1, p_2, \ldots, p_k)$, instead of working with a matrix A of large integers, we may work with the collection of residues $(A \bmod p_i)$, for each $p_i \in \mathcal{P}$. Informally, the Chinese Remainder Theorem (CRT) states that an integer x with $0 \leq x < b$ can be uniquely represented with respect to $\mathcal{P}$ as $((x \bmod p_1), (x \bmod p_2), \ldots (x \bmod p_k))$ when $\prod_{i=1}^{k} p_i = P > b$. Slightly more formally, the CRT establishes an isomorphism between $\mathbb{Z}_P$ and $\mathbb{Z}_{p_1} \times \mathbb{Z}_{p_2} \times \ldots \times \mathbb{Z}_{p_k}$.

The following is a simple example of the technique applied to computing the product of two 2×2 matrices with $\mathcal{P} = [11, 13, 17]$. Each operand is reduced modulo each element of basis $\mathcal{P}$ (first two rows below). The two resulting collections of residues can then be multiplied element-wise modulo the appropriate modulus p_i (last three columns below). Finally, the residues corresponding to the product can be reconstructed by means of the Chinese remainder algorithm to obtain the desired product.

		mod 11	mod 13	mod 17
A	$\begin{bmatrix} 25 & 21 \\ 16 & 18 \end{bmatrix}$	$\begin{bmatrix} 3 & 10 \\ 5 & 7 \end{bmatrix}$	$\begin{bmatrix} 12 & 8 \\ 3 & 5 \end{bmatrix}$	$\begin{bmatrix} 8 & 4 \\ 16 & 1 \end{bmatrix}$
B	$\begin{bmatrix} 32 & 23 \\ 13 & 29 \end{bmatrix}$	$\begin{bmatrix} 10 & 1 \\ 2 & 7 \end{bmatrix}$	$\begin{bmatrix} 6 & 10 \\ 0 & 3 \end{bmatrix}$	$\begin{bmatrix} 15 & 6 \\ 13 & 12 \end{bmatrix}$
$A \cdot B$	$\begin{bmatrix} 1073 & 1184 \\ 746 & 890 \end{bmatrix}$	$\begin{bmatrix} 6 & 7 \\ 9 & 10 \end{bmatrix}$	$\begin{bmatrix} 7 & 1 \\ 5 & 6 \end{bmatrix}$	$\begin{bmatrix} 2 & 11 \\ 15 & 6 \end{bmatrix}$

The primes composing the basis of a residue number system are typically chosen to be "word-sized" — the size of the operands of the machine's native arithmetic routines — but as large as possible. Larger primes imply fewer total residues and, thus, less work overall. On modern machines, this suggests 32- or 64-bit primes; however, the nature of ATLAS BLAS imposes an additional restriction. As ATLAS operates on matrices of double-precision (i.e., 64-bit) floating point values while the matrices at hand are conceptually integral, we require each modulus be chosen such that the entries of the matrix are representable exactly. The standard representation of a double-precision number can exactly represent integers less than 2^{53}: 12 bits are used for the sign and exponent, leaving only 52 for the mantissa (one bit of precision is implicitly represented). To maintain exact values at all stages of the computation, the product of any two matrices must meet this bound. Specifically, each modulus p_i must satisfy $n(p_i - 1)^2 \leq 2^{53} - 1$.

In practice, the largest prime in each basis is selected to be the largest prime p_k satisfying

$$p_k < \frac{\sqrt{4n(2^{53} - 1) + 1} + (2n - 1)}{2n}.$$

The remaining moduli are selected in decreasing order from this initial value.

4.2 High-order residue implementation concerns

As presented, `DoublePlusOneLift` requires X and k (respectively, the base of the X-adic lifting and the number of lifts) as input. For purposes of this implementation, X and k were selected to be sufficiently large as required by Theorem 7 and Corollary 9.

In addition to ATLAS for matrix operations, we make limited use of the Integer Matrix Library (IML) [6] and the GNU Multi-Precision Arithmetic (GMP) library [15] for matrix

inversion over a finite field and big integer arithmetic, respectively. Recall the former is required for computing the initial matrix inverse with respect to the lifting modulus; the latter is used for, among other small applications, representing input and output matrices.

4.2.1 Basis conversion in a residue number system

The lifting steps of the high-order residue algorithm require both operations on arbitrarily large integers (i.e., conceptually over $\mathbb{Z}$) and operations with respect to a possibly large lifting modulus (i.e., over $\mathbb{Z}_X$). One basis $\mathcal{P}$ may be used to represent the integer operations: provided $p_1 p_2 \ldots p_k \geq 1.2002n||A||$ as per the bound for R_i in Theorem 7. However, an additional basis $\mathcal{Q}$ is required to represent operations modulo X by selecting $X = q_1 q_2 \ldots q_k$. The following lines are excerpted from Algorithm 4 (lines 5 and 6).

$M_i := \text{Rem}(B_0 \overline{R}, X)$

$R_{i+1} := (1/X)(\overline{R} - AM_i)$

Note that M_i is computed over $\mathbb{Z}_X$, but used in a computation over $\mathbb{Z}$ in the following line. Computationally, M_i is computed with respect to basis $\mathcal{Q}$ in the first line and appears in a computation with respect to $\mathcal{P}$ in the second. Moreover, $\mathcal{Q}$ must be distinct from $\mathcal{P}$ as X is required to be invertible with respect to basis $\mathcal{P}$. As a consequence of these requirements, an algorithm for converting between residues in the two coprime residue number systems is necessary. There are several possibilities.

Notation

For conciseness, we use subscripted square brackets to denote modular reduction: $[x]_d := (x \bmod d)$. So for $A \in \mathbb{Z}^{n \times n}$, we denote the ith modular image with respect to basis $\mathcal{P}$ — i.e., $A \bmod p_i$ — as $[A]_{p_i}$. The goal, then, is to move from a given representation of A, $\left[[A]_{p_1}, [A]_{p_2}, \ldots, [A]_{p_k}\right]$, in basis $\mathcal{P}$ to an equivalent representation in basis $\mathcal{Q}$, $\left[[A]_{q_1}, [A]_{q_2}, \ldots, [A]_{q_k}\right]$.

The following quantities are useful throughout this section. Note that each relies only on the basis of the residue number system, not the value being represented. Therefore, each of these quantities below can be precomputed at the start of the algorithm and used

throughout the remainder at no cost.

$$\begin{aligned}
\mathcal{P} &= [p_1, p_2, p_3, \ldots, p_k] \\
p &= \prod_i p_i \\
\ell_{p_i} &= \prod_j p_j/p_i \\
s_{p_i} &= \ell_{p_i}^{-1} \bmod p_i \\
c_{p_i} &= \ell_{p_i} s_{p_i} \\
e_{p_i} &= s_{p_i}/p_i
\end{aligned}$$

The equivalent quantities corresponding to basis $\mathcal{Q}$ are denoted analogously in the obvious way.

Reconstruction

A straightforward approach is to reconstruct the integer matrix A using the standard Chinese Remainder Algorithm. Once matrix A is in hand, each of residues in the new basis (each $[A]_{q_i}$) can be computed directly from A. Using the quantities and notation defined above, this method can be summarized as follows.

$$[A]_{q_i} = \left[\left[\sum_{j=1}^{k} c_{p_j} [A]_{p_j} \right]_p \right]_{q_i}$$

Note that although each residue $[A]_{q_i}$ produced by the above is bounded by q_i, the intermediate values may be (and, in general, are) much larger. Specifically, the inner sum above must be computed to full precision using arbitrary-sized integer arithmetic. Neither keeping the running sum reduced modulo p nor modulo q_i computes the correct result.

Redundant Modulus

The standard presentation of the Chinese Remainder Theorem gives an expression equivalent to A only in a modular sense:

$$A \equiv \sum_{j=1}^{k} c_{p_j} [A]_{p_j} \bmod p.$$

An alternative presentation, however, relates the quantities exactly in terms of an unknown error term R.

$$A = \sum_{j=1}^{k} c_{p_j} [A]_{p_j} - pR$$

Then, deferring the issue of computing R, the target residue $[A]_{q_i}$ can be computed using only arithmetic modulo q_i.

$$[A]_{q_i} = \left[\left[\sum_{j=1}^{k} [c_{p_j} [A]_{p_j}]_{q_i} \right]_{q_i} - [pR]_{q_i} \right]_{q_i}$$

Both Shenoy/Kumaresan [32] and Posch/Posch [31] use this formulation but present two different methods for computing the correction term R. The approach of Shenoy and Kumaresan relies on having access to an extra independent modulus and corresponding residue. The requirement that the extra residue be obtained independently renders this approach unsuitable for our purposes.

Posch

The method of Posch and Posch[31], however, is suitable. Beginning with the Chinese remainder equation with error term R, dividing each term by $p = \prod_i p_i$ yields the following.

$$A = \sum_{j=1}^{k} c_{p_j} [A]_{p_j} - pR$$

$$R + A/p = \sum_{j=1}^{k} e_{p_j} [A]_{p_j}$$

As A is reduced modulo p, A/p is a rational number less than one; thus, the second line above gives an approximate expression for R. So, computing $R^* = \lfloor \sum_{j=1}^{k} e_{p_j} [A]_{p_j} \rfloor$ gives either R or $R - 1$. Distinguishing between the two cases is difficult, though Posch and Posch do describe a method to do so. They additionally provide a bound on the error

in R^*: for our purposes, $\epsilon < 1/p_{\max}$. Thus, R can be computed directly as follows.

$$R = \lfloor \sum_{j=1}^{k} [A]_{p_j} e_{p_j} \rceil$$

$$[A]_{q_i} = \left[\left[\sum_{j=1}^{k} [c_{p_j} [A]_{p_j}]_{q_i} \right]_{q_i} - [pR]_{q_i} \right]_{q_i}$$

Once the error term R is known, the computation of each target residue can be performed modulo q_i, rather than requiring reconstruction to full precision.

Summary

The method of Posch and Posch avoids the need for reconstruction and works well with residue number systems having fewer moduli. Standard reconstruction works best, however, with larger bases. While the later method passes over the source residues once (at the expense of having to deal with ever increasing operands), the former requires passing over the source residues once for each residue in the target basis. By default, the implementation uses the simple reconstruction method; adaptively selecting between these two methods may improve performance of the implementation slightly.

4.3 Hermite normal form implementation concerns

Our implementation relies on several existing, highly efficient libraries. Nonsingular system solving is provided by the Integer Matrix Library (IML) [6]. Additional routines not available elsewhere (Algorithm 6 for computing a minimal triangular denominator, for one) are implemented in terms of the integer arithmetic routines of the GNU Multi-Precision Arithmetic (GMP) library [15].

4.3.1 Projection size

The column dimension k of the random $v \in \mathbb{Z}^{n \times k}$ used to compute the projection $x := A^{-1}v$ (cf. the "Projection" phase in figure 3.2) may be varied from one iteration to the next. Choosing a value for k, the size of the projection, is an inexact process driven mostly by empirical observation. A reality of the lifting-based linear system solving algorithms, like

those in IML, is that the cost of initialization can meet or surpass the cost of the lifting steps themselves. Indeed, to cover a wide range of inputs, IML has been tuned to balance the cost of initialization with the cost of lifting. The relevant consequence here, then, is that the cost of computing a projection of multiple columns is negligibly more than the cost of working with a single column.

Preliminary observations suggest that an initial projection of eight columns works well. Eight columns are sufficient to capture all invariant factors in the case of a random matrix without being prohibitively more costly than working with a single column. Using fewer columns yields almost no time savings, but increases the likelihood of further projections being required. If more than eight columns are used, the extra computation time grows to more than a few percent. This choice is essentially arbitrary and determined only by a highly unscientific process of trial-and-error.

Subsequent iterations can use much larger projections as the largest invariant factors will have already been extracted and, consequently, the system can be solved at a much lower precision: this entry size/dimension trade-off is described further in Section 3.2. The scheme used in this implementation is somewhat coarse, but effective. The second iteration uses a projection of $n/10$ columns; the third iteration uses a projection of n columns. Additionally, rather than a randomly chosen matrix, the third iteration uses the identity matrix and thus computes A^{-1} exactly. This guarantees that only three iterations are ever required and obviates a final high-order residue computation to certify the result.

The choice of three iterations and of $n/10$ columns for the second projection only appear to work well and no claim is made as to their optimality. It is perhaps possible to choose the size of projections adaptively, based on the size of the denominator of the previous projection (or, even better, based on the expected size of the next one). If the previous denominator is larger than expected, the next denominator may be relatively small and, thus, the next projection could be made larger without incurring much additional cost. An adaptive scheme of this sort would require quantifying, perhaps only in a heuristic sense, the expected size of the invariant factors extracted at each iteration.

4.3.2 Packed triangular representation

As each projection yields a portion of the Hermite form corresponding to only a few invariant factors, combining these "slices" involves operations on highly structured matrices. Generally, the non-zero elements of these matrices are confined to only a few columns. Working with these matrices as if they were generic matrices introduces a significant inefficiency.

Lines 3 and 7 in Algorithm 7 are problematic for large k. Each iteration requires a matrix/vector product as well as a matrix/matrix product. For $k \approx n$, this entire process would appear to require at least n^4 integer operations; this is too costly by a factor of n. In the worst conceivable case, this rough initial analysis holds. However, the matrices arising from `hcol` (T, say) in practice exhibit properties rendering this approach tenable. Specifically, many of the diagonal elements of T are 1. As T is in Hermite normal form, all off-diagonal entries in these columns are zero; these are said to be "trivial columns". All other columns (i.e., those with non-unit diagonal entries) are said to be "non-trivial columns".

Again, this property does not necessarily apply: a given invariant factor may split into any number of non-trivial columns of the Hermite form. Yet, excepting matrices explicitly constructed to be degenerate in this sense, any implementation can make a large practical improvement by leveraging this particular form of sparsity. Consider a matrix with the structure shown below. Here, only columns 2, 3, and 6 are non-trivial. Storing only the non-trivial columns and their positions can immediately reduce the amount of space required to store the matrix. Matrices in this form (e.g., the right matrix below) are said to be "packed".

$$\begin{bmatrix} 1 & * & * & & & * \\ & * & * & & & * \\ & & * & & & * \\ & & & 1 & & * \\ & & & & 1 & * \\ & & & & & * \end{bmatrix} \longrightarrow \left[\begin{array}{ccc} * & * & * \\ * & * & * \\ & * & * \\ & & * \\ & & * \\ & & * \\ \hline 2 & 3 & 6 \end{array}\right]$$

Moreover, the necessary matrix operations can be performed on packed matrices, considering only the non-trivial columns. For a packed matrix Z, `cols(Z)` denotes the set of non-trivial columns; in the example above, $\texttt{cols(Z)} = \{2, 3, 6\}$.

Packed Matrix Multiplication

The product of two packed matrices $Z = ST$ can be computed with a careful modification of the classical iterative definition of matrix multiplication.

The set of non-trivial column indices in the product is the union of non-trivial column indices in each of the operands: $\texttt{cols(Z)} := \texttt{cols(S)} \cup \texttt{cols(T)}$. That is, any non-trivial column occurring in the result must correspond to non-trivial columns in one of the inputs. Then, for row r ranging from 1 to n and column $c \in \texttt{cols(Z)}$, the corresponding entry in

Z is defined as follows.

$$Z[r,c] = \begin{cases} \sum_{i \in \texttt{cols(S)}} S[r,i]T[i,c] & c \in \texttt{cols}(S) \wedge c \in \texttt{cols}(T) \\ \sum_{i \in \texttt{cols(S)}} S[r,i]T[i,c] + T[r,c] & c \notin \texttt{cols}(S) \wedge c \in \texttt{cols}(T) \\ S[r,c] & c \in \texttt{cols}(S) \wedge c \notin \texttt{cols}(T) \end{cases}$$

Note that the first case is exactly classical matrix multiplication. The second case is similar, but accounts for the implicit 1 in column c of S. In all cases, an entry is referenced only when the corresponding column is non-trivial: that is, $Y[*,x]$ is referenced only when $x \in \texttt{cols(Y)}$. Algorithm 8 gives the complete algorithm for packed matrix multiplication.

If S and T have $k < n$ non-trivial columns, Algorithm 8 requires $\Theta(nk^2)$ integer operations. For k much smaller than n, this is an obvious improvement over the $O(n^3)$ operations required in standard matrix multiplication.

Algorithm 8 `packedMatrixMultiply`(S,T) (iterative)

Input: Packed matrices $S, T \in \mathbb{Z}^{n \times n}$.
Output: $Z := ST$.

1: $Z := I_n$
2: $\texttt{cols}(Z) := \texttt{cols}(S) \cup \texttt{cols}(T)$
3: **for** $row = n$ **to** 1 **do**
4: **for** col **in** $\texttt{cols}(Z)$ **do**
5: $$Z[r,c] = \begin{cases} \sum_{i \in \texttt{cols(S)}} S[r,i]T[i,c] & c \in \texttt{cols}(S) \wedge c \in \texttt{cols}(T) \\ \sum_{i \in \texttt{cols(S)}} S[r,i]T[i,c] + T[r,c] & c \notin \texttt{cols}(S) \wedge c \in \texttt{cols}(T) \\ S[r,c] & c \in \texttt{cols}(S) \wedge c \notin \texttt{cols}(T) \end{cases}$$
6: **end for**
7: **end for**
8: **return** Z

Much of the packed algorithm simply rephrases the internals of classical matrix multiplication. For sufficiently large matrices, it is beneficial to directly use existing dense multiplication implementations to compute as much of the product as possible. Algorithm 9 first performs a standard multiplication of the columns of S with the appropriately corresponding rows of T before applying the necessary corrections.

Specifically, a dense matrix A is formed from the non-trivial columns of S. Likewise, a dense matrix B is formed from the *rows* of T corresponding to the non-trivial columns of S. A large portion of $Z = ST$ then appears in the dense product $C = AB$: this product may

be computed by any available efficient method for dense matrix multiplication (ATLAS is used in this implementation). If S and T have k_S and k_T non-trivial columns, respectively, the dense multiplication is between matrices of dimension of $n \times k_S$ and $k_S \times k_T$. The resulting matrix C (dimension $n \times k_T$) form the columns in Z as follows.

$$Z[*,c] = \begin{cases} C[*,c] & c \in \texttt{cols}(S) \wedge c \in \texttt{cols}(T) \\ C[*,c] + T[*,c] & c \notin \texttt{cols}(S) \wedge c \in \texttt{cols}(T) \\ S[*,c] & c \in \texttt{cols}(S) \wedge c \notin \texttt{cols}(T) \end{cases}$$

Here, the correction specified in the section case corresponds to the implicit identity columns in S; likewise, the third case corresponds to the identity columns of T. Note that all necessary multiplications are performed by the initial matrix multiplication. Additionally, only the second case requires arithmetic of any sort. The other cases reference values previously known or computed and thus the bulk of the computation is performed by the single dense matrix multiplication.

Algorithm 9 `packedMatrixMultiply`(S,T) (block)

Input: Packed matrices $S, T \in \mathbb{Z}^{n \times n}$.
Output: $Z := ST$.

1: **for** $\hat{c} = 1$ **to** $|\texttt{cols}(S)|$ **do**
2: $\quad c = \texttt{cols}(S)[\hat{c}]$
3: $\quad$ **for** $i = 1$ **to** n **do**
4: $\quad\quad A[i, \hat{c}] := S\big[i, c\big]$
5: $\quad\quad B[\hat{c}, i] := T[c, j]$
6: $\quad$ **end for**
7: **end for**
8: $C = AB$ {dense matrix multiply}
9: **for** c **in** $\texttt{cols}(Z)$ **do**
10: $\quad$ **for** $r = 1$ **to** n **do**
11: $\quad\quad Z[r,c] := \begin{cases} C[r,c] & c \in \texttt{cols}(S) \wedge c \in \texttt{cols}(T) \\ C[r,c] + T[r,c] & c \notin \texttt{cols}(S) \wedge c \in \texttt{cols}(T) \\ S[r,c] & c \in \texttt{cols}(S) \wedge c \notin \texttt{cols}(T) \end{cases}$
12: $\quad$ **end for**
13: **end for**
14: **return** Z

The blocked method incurs some overhead in the construction of dense matrices A and B. For matrices with very few non-trivial columns, then, the iterative method is best;

the blocked method performs best as the number of non-trivial columns increases. Rough empirical tests indicate the crossover point is approximately 40 non-trivial columns. If both S and T have more than 40 non-trivial columns, our implementation uses the blocked method.

4.3.3 Balanced multi-column minimal triangular denominator

The following excerpt from Algorithm 7 highlights a further performance issue. Computing the minimal triangular denominator corresponding to a multiple-column projection, if done naïvely, requires a series of operations, each of which is fast, but that may incur a significant amount of overhead in aggregate.

for $i = 1$ **to** k **do**
  $y := H.X[i] \bmod d$
  $\cdots$
  $T := \texttt{hcol}(y, d')$
  $H := T.H$
od

For each column in the projection, a single minimal triangular denominator T is computed. Each such T is left-multiplied with the product of the previous to obtain H, a minimal triangular denominator for the first columns of X. The next column in the projection is then left multiplied by H. In informal terms, each column captures some portion of the Hermite normal form, working instead with Hx on subsequent columns removes this already-captured portion. This approach requires k multiplications between an $n \times n$ matrix and a single column vector. Each multiplication is relatively cheap, but the total work performed is split too finely over many operations. It would be desirable to collect multiple matrix-vector products into a single larger matrix-matrix product.

An equivalent approach applies each T to the remaining columns in the projection. Here, the dimensions of the operands are as balanced as possible, but each T has few non-trivial columns. Considering the packed matrix routines of the previous section, the situation is essentially unchanged: many operations on narrow matrices/vectors.

A better solution balances the application of T to X and the combination of previously computed Ts into H. For the purposes of this discussion, we refer to each minimal triangular denominator (and products thereof) as a "factor"; a factor obtained from a single application of `hcol` (and not a product of factors) is said to be a single-column factor. The "size" of a factor is said to be the number of single-column factors composing the factor in question. (The product of two single-column factors is a factor of size two; the product

of a factor of size two and a single-column factor is a factor of size three.) We maintain an ordered collection of factors, combining them (i.e., multiplying them together) only when two consecutive factors are of the same size. Upon its creation, a factor of size s is applied to the next s columns in the projection. An simple example is illustrative.

Consider a projection X of 8 columns x_1 through x_8; the collection of factors is denoted S.

$$S = \emptyset$$
$$X = \left[\begin{array}{cccccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \end{array} \right]$$

The first (single-column) factor T_1 is obtained from first column x_1; no other factors exist, so T_1 is applied to the next column, x_2.

$$S = \{T_1\}$$
$$X = \left[\begin{array}{cccccccc} * & T_1x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \end{array} \right]$$

Factor T_2 is then obtained from the second column (now T_1x_2). As the collection of factors is now $\{T_2, T_1\}$ and these two factors are the same size, they are combined: $T_{21} := T_2T_1$. The resulting factor, T_{21} of size 2, is then applied to the next two projection columns.

$$S = \{T_{21}\}$$
$$X = \left[\begin{array}{cccccccc} * & * & T_{21}x_3 & T_{21}x_4 & x_5 & x_6 & x_7 & x_8 \end{array} \right]$$

The third column yields T_3, which is applied to the fourth column.

$$S = \{T_3,\ T_{21}\}$$
$$X = \left[\begin{array}{cccccccc} * & * & * & T_3T_{21}x_4 & x_5 & x_6 & x_7 & x_8 \end{array} \right]$$

In turn, the fourth column yields T_4. The collection S is now $\{T_4, T_3, T_{21}\}$. The component factors are then combined recursively: $T_{43} := T_4T_3$ and $T_{4321} := T_{43}T_{21}$. The result of this cascading combination is T_{4321}, a factor of size 4, which is thus applied to the next four columns.

$$S = \{T_{4321}\}$$
$$X = \left[\begin{array}{cccccccc} * & * & * & * & T_{4321}x_4 & T_{4321}x_5 & T_{4321}x_6 & T_{4321}x_7 \end{array} \right]$$

This process continues until all the columns of X have been processed. The algorithm can then return $T_{87654321}$, a minimal triangular denominator for the entire projection.

Keeping the operands balanced results in fewer matrix/matrix multiplications overall and largely avoids unbalanced multiplications. The balanced method has the same asymptotic complexity (after all, the same total work is being done in a different order) but results in a practical gain. For a matrix of dimension 400 constructed to have a highly non-trivial Hermite form, computing the minimal triangular denominator for a projection of size 400 requires 11.53 seconds with the naïve method, and only 4.11 seconds with the balanced method. As expected, the gain of the balanced method is larger for larger matrices and projections. In fact, the inefficiency of the naïve method causes its cost to surpass the costs of the high-order reside computation and the linear system solve for input matrices of dimension as small as 8000.

Algorithm 10 gives the entirety of the balanced algorithm.

4.4 Empirical results

The following section contains a selection of our experimental results, as well as a discussion of our methodology and of the results themselves. As there is no obvious comparable for the high-order residue computation, the results presented are purely for completeness and the discussion purely conversational. There do exist, however, exceptional implementations of algorithms for Hermite normal form in computer algebra systems. We compare our implementation against existing implementations on range of varying inputs: our implementation performs very well.

4.4.1 High-order residue

Tables 4.1 and 4.2 summarize our experimental results. These timings were made on an Intel 1.3 GHz Itanium2 with 192 GB RAM running GNU/Linux 2.4.21. The software was compiled with gcc 4.1.2 and linked against IML 1.0.3, ATLAS 3.6.0, and GMP 4.1.3. Tests were performed on randomly generated input matrices of two types: those with single decimal digit entries and those with 100 decimal digit entries.

The published timings[1] for linear system solving with IML were performed on a very similar machine and provide a point of comparison. Using IML, solving a linear system

[1] http://www.cs.uwaterloo.ca/~astorjoh/iml.html

Algorithm 10 `hermiteOfProjection_balanced`(X, d)

`collapseAll`(S)

1: $H := I_n$
2: **for** $i = 1$ **to length**(S) **do**
3: $\quad (h, c) := S[i]$
4: $\quad H := hH$
5: **end for**
6: **return** H

`collapseLast`(S)

1: **while** `length`$(S) >= 2$ **do**
2: $\quad (h_1, c_1), (h_2, c_2) = S[-1], S[-2]$
3: $\quad$ **if** $c_1 \neq c_2$ **then break**
4: $\quad h := h_1 h_2$; $c := c_1 + c_2$
5: $\quad$ **pop** $S[-1], S[-2]$ from S
6: $\quad$ **append** (h, c) to S
7: **end while**

`hermiteOfProjection_balanced`(X, d)

Input: $X \in \mathbb{Z}^{n \times n}$ and $d \in \mathbb{Z}$.
Output: $H \in \mathbb{Z}^{n \times n}$, a minimal triangular denominator of X/d.

1: $S := [(I_n, 0)]$;
2: **for** $i = 1$ **to** 1 **do**
3: $\quad (h, c) := S[-1]$;
4: $\quad X[*, i..i + c] := h.X[*, i..i + c] \bmod d$
5: $\quad y := X[*, i]$;
6: $\quad g := \mathrm{Gcd}(y); d' := d/g; y := y/g$;
7: $\quad T :=$ `hcol`(y, d');
8: $\quad$ **append** $(T, 1)$ to S
9: $\quad$ `collapseLast`(S);
10: **end for**
11: $H :=$ `collapseAll`(S);
12: $H :=$ `triangularHNF`(H);
13: **return** H

Dimension	Time	
1000	57 s	
2000	454 s	($\approx$ 7.6 minutes)
4000	3756 s	($\approx$ 62.6 minutes)
8000	41120 s	($\approx$ 11.4 hours)

Table 4.1: Time to compute high-order residue: 1 decimal digit entries.

Dimension	Time	
200	5 s	
400	33 s	
1000	472 s	($\approx$ 8 minutes)
2000	4336 s	($\approx$ 1.2 hours)

Table 4.2: Time to compute high-order residue: 100 decimal digit entries.

of dimension 2000 with 100-digit entries required about 1.3 hours. Here, computing the sparse inverse expansion with input of the same size required approximately the same amount of time. IML timings in P. Giorgi's dissertation [13] are also comparable: for input of dimension 2000 with 30 digit entries, solving a linear system and computing a sparse inverse expansion both require about thirty minutes. This result suggests (at least for input of about this size) that high-order lifting has some potential as a practical approach to the problems of, for instance, determinant calculation or integrality certification.

Dimension	Input size (MB)	Peak usage (MB)
200	2.56	51.52
400	10.24	212
1000	64	1360
2000	256	5600

Table 4.3: High-order residue emory usage with 100 decimal digit input entries.

Tables 4.3 and 4.4 show the memory usage of selected computations. The majority of the extra space is used to store intermediate computations in the residue number systems. That is, the extra space required is proportional to the number of elements in the two coprime bases and, consequently, exceeds the size of the input by only a logarithmic factor.

Dimension	Input size (MB)	Peak usage (MB)
1000	24	208
2000	96	832
4000	384	3332
8000	1536	13312

Table 4.4: High-order residue memory usage with 1 decimal digit input entries.

4.4.2 Hermite normal form

Although the Hermite normal form algorithm of the previous section is not asymptotically optimal, empirical tests with a careful implementation bear out its effectiveness in practice.

The projection-based method for extracting the invariant structure is sensitive to the number of non-trivial invariant factors. A matrix with many non-trivial invariant factors requires the computation of many projections. However, as generic matrices are expected to have very few non-trivial invariant factors - and often only one - the practical performance of the algorithm in the generic case is very good. Typically, only a single projection is required to extract the entirety of the invariant structure.

Yet, while the projection method performs most dramatically on generic matrices, the algorithm can also be effectively applied to matrices specifically constructed to have many non-trivial invariant factors and, in turn, highly non-trivial Hermite and Smith forms. This implementation is robust in its ability to handle all input matrices without making undue concessions to either the common or exceptional cases. No special manual tuning is used to better handle any particular case.

Methodology

Three classes of matrices are used to illustrate the performance of the implementation at both extremes of the spectrum of input matrices.

Firstly, to test the implementation on the generic case, we use matrices with random independent, identically distributed entries selected uniformly in the range 0 through $2^{\ell}-1$, for various values of the bit lenght ℓ. These matrices commonly have only a single non-trivial column in their Hermite form and very rarely have more than few. For these random matrices, the implements performs best as only a single projection is required.

Following the example of Jäger and Wagner [19], we use the following class of matrices to test performance on more difficult inputs with many non-trivial invariant factors:

$$\mathcal{J}_n = (\jmath_{i,j}) \text{ with } \jmath_{i,j} = (i-1)^{j-1} \bmod n \text{ for } 1 \leq i, j \leq n$$

If n is prime, $\mathcal{J}_n$ is nonsingular[2], typically has more than $n/2$ non-trivial invariant factors, and s_n is very large relative to n. For instance, $\mathcal{J}_{113}$ has 72 non-trivial invariant factors, the largest of which is 253 bits in length. In addition to having the desired structural properties, $\mathcal{J}_n$ can be quickly and straightforwardly constructed, allowing for consistent comparisons between implementations.

Finally, we use the class of matrices described by A. Steel on his "Hermite Normal Form Timings Page"[33]. Beginning with a diagonal matrix of random integers between 1 and $n/10$, a random row is added/subtracted from a second random row. This process repeats $n/10$ times, and is followed with $n/10$ analogous column operations. Typically, the resulting matrices have entries fewer than 20 bits in size, with an average entry size of about 13 bits. Like the Jäger class matrices, the Steel class matrices have Hermite forms with approximately $n/2$ non-trivial columns. In contrast, however, the largest diagonal entry is much smaller and all diagonal entries grow smoothly; Jäger-style matrices may have very large diagonal entries and vary widely in size.

The relevant differences in the three types of input above are best described by considering the diagonal entries of their Hermite normal forms. The examples below show the diagonal Hermite entries corresponding to matrices of each type, all of dimension 53; note the differences in the number and size of the non-trivial entries.

Random matrix, 8-bit entries:

$$\overbrace{1, \ldots, 1}^{51 \text{ ones}}, 5, 223533630063866909 \overbrace{\cdots \cdots}^{100 \text{ more digits}} 32104719891529157$$

Jäger class:

$$\overbrace{1, \ldots, 1}^{18 \text{ ones}}, 2, 1, 1, 1, 4, 1, 4, 1, 1, 2, 2, 4, 4, 4, 4, 4, 4, 4, 8,$$
$$4, 8, 1428, 4, 4, 4, 4, 32, 32, 16, 28, 231243877351960928, 4038944,$$
$$41392654046001006112, 1283400845611588095632089246604480, 73032$$

[2]For prime n, $\mathcal{J}_n$ is nonsingular as it is a Vandermonde matrix.

Steel class:

$$\overbrace{1,\ldots,1}^{32\text{ ones}},2,2,1,2,2,2,10,10,10,10,10,60,30,60,60,60,60,60,60,60,60$$

The preëminent practical algorithms for integer Hermite normal form are provided by computer algebra systems Sage [35] and Magma [5] and provide excellent points of comparison. Like our algorithm, Sage's Hermite normal form algorithm [30] and implementation rely on the fast linear system solving provided by IML and are designed to be most effective in the case of random matrices. In the case of random matrices with entries "at all large", Sage's implementation is claimed to be "fastest in the world available anywhere [*sic*]" [34]. Magma's algorithm is described only as "modular" [33] and, based on the empirical timings, presumably shares some features with both our algorithm and Sage's; as Magma's source is closed, this comment is purely speculative but not wildly so.

The experiments were performed on an 64-bit AMD Opteron 8356 at 2.3 GHz. The software was compiled with GCC 4.6.3 and linked against IML 1.0.3, ATLAS 3.8.4, and GMP 5.1.1. Comparisons are made with Sage 5.2 and Magma 2.19.

Results

The following tables summarize the experimental results.

Table 4.5 highlights performance results for random matrices. All three implementations perform very well on these types of inputs. When entries are small, Magma holds a slight (but constant) performance edge with small entries, but quickly degrades as the entry size becomes even moderately large. For larger entry sizes, our implementation holds a slight advantage over Sage. For very large entries (more than 512 bits, say), the gap between Sage and our implementation narrows until the two are essentially at parity. Excepting perhaps Magma with large entries, no implementation is at an asymptotic disadvantage; across the board, as input dimension doubles, the time required increases roughly eightfold.

Steel-class matrices are more difficult; Table 4.6 summarizes the results. Here, Magma outperforms our implementation, but again, only by a constant factor. Sage's implementation does not handle non-generic matrices well (nor does it profess to); its strength is intended to be the case of random inputs. Sage excluded, running times again appear to grow as $O^{\sim}(n^3)$.

Finally, Jäger-class matrices have the most involved Hermite form and thus provide a greater challenge: Table 4.7 shows these results. On these types of input, neither Sage nor

entry size	n	time[3] (s) this	Magma 2.19	Sage 5.2
8 bits	500	7.57	**6.00**	21.19
	1000	51.73	**48.23**	139.98
	2000	398.40	**370.73**	1013.93
32 bits	500	**21.71**	28.68	33.02
	1000	**148.72**	238.39	226.57
	2000	**1144.75**	1739.44	1808.69
64 bits	200	**4.03**	39.03	5.79
	400	**23.64**	320.16	31.98
	800	**147.64**	2823.35	198.35

Table 4.5: HNF comparison — random matrices.

n	this	Magma 2.19	Sage 5.2
100	**0.150**	0.330	2.01
200	3.67	**2.12**	31.39
400	19.05	**14.03**	480.9
800	124.77	**97.69**	
1000	229.93	**196.72**	

Table 4.6: HNF comparison — Steel-style matrix.

Magma performs well; the performance of both degrades very quickly and even moderately size instances are intractable. Presumably, Magma's implementation is able to take advantage of the small size of the Hermite diagonal in the case of the Steel-class matrices and this advantage is lost on the much larger entries in the Hermite forms of Jäger-class matrices. For these inputs, our implementation is decisively fastest and, again appears to behave as $O\tilde{}(n^3)$.

n	this	Magma 2.19	Sage 5.2
101	**0.52**	1.98	2.29
211	**2.98**	44.17	38.06
401	**20.54**	1528	912.9
809	**123.6**		
1009	**232.1**		

Table 4.7: HNF comparison — Jäger-style matrix.

The above results show that while Magma and Sage both hold narrow advantages on certain types of input, they perform quite poorly on others. In contrast, our implementation is competitive across all inputs and appears to admit no input for which the computation time is critically bad. Even on matrices with highly non-trivial Hermite forms, our implementation's running time grows as $O\tilde{}(n^3)$.

Focussing on the performance of our implementation, the following results attempt to bolster the case for our claimed $O(n^3 \log n)$ running time.

n	time (s)	n	time (s)
100	0.09	125	0.14
200	0.42	250	0.75
400	3.08	500	5.66
800	22.5	1000	42.0
1600	171	2000	348
3200	1625	4000	3214

Table 4.8: Time to compute Hermite form of random $n \times n$ matrix with 8-bit entries.

For random matrices (Table 4.8), the computation time grows roughly as $n^3 \log n$. That is, doubling the input dimension increases the cost by a factor of slightly less than nine. The fit is not perfect, however: smaller inputs slightly outperform expectations while larger

inputs are slightly under-performing. For instance, the ratio between results for $n = 4000$ and $n = 2000$ is greater than nine, but is near seven between $n = 1000$ and $n = 500$. Smaller input matrices may be taking advantage of some beneficial machine-specific cache effects.

n	k	time (s)	n	k	time (s)
101	56	0.52	127	77	0.83
211	118	2.91	251	132	5.90
401	266	18.6	503	252	32.0
809	503	118	1009	663	221
1601	1060	831	2003	1041	1410

Table 4.9: Time to compute Hermite form of $\mathcal{J}_n$ with k non-trivial invariant factors.

For inputs with many non-trivial invariant factors (Table 4.9), the empirical timings again grow roughly as $n^3 \log n$. Although the algorithm runs much faster overall on generic inputs, the rate of growth exhibited by the timings is the same for both types of inputs.

l	n	
	400	800
3	2.28	16.61
8	3.79	27.14
16	6.40	44.63
32	11.42	78.56
48	16.72	116.47
64	23.64	147.64
80	31.08	199.13
96	37.35	244.17
128	57.70	369.60

Table 4.10: Time to compute Hermite form of random matrix of with l-bit entries.

Finally, Table 4.10 considers an second aspect of input size. If the dimension is held constant, but the entry size is varied, the computation time increases roughly linearly. This trend is somewhat diminished in the largest sizes: large entries perform slightly worse than would be expected.

4.5 Concluding remarks

Representing arbitrarily-sized integer matrices over a residue number system allows an efficient implementation of double-plus-one lifting in terms of word-sized arithmetic and practically fast matrix multiplication provided by optimized implementations of the level 3 BLAS. The additional complication of moving between residue number systems representing operations over $\mathbb{Z}$ and over $\mathbb{Z}_X$ is resolved by a pair of methods for basis conversions. Timings show the time required to computing a high-order residue as roughly equivalent to the time required to solve a linear system of comparable size.

A set of effective optimizations can leverage blocked operations and minimize overhead whenever possible; the result is a practically very fast implementation of our Hermite normal form algorithm. Empirical experiments show our implementation, no matter the type of input, is consistently competitive with and often bests the fastest-known implementations.

Many of the optimization techniques used have a limited range of applicability. An interesting and likely fruitful meta-optimization would be the development of more advanced heuristics for choosing between methods (between blocked and iterative approaches, say). Where the current implementation has any notion of such thresholds at all, they are often coarse and were hastily decided upon.

Although the Hermite form implementation also naturally reveals the determinant (multiply the diagonal entries), a properly optimized implementation of our determinant algorithm is as yet unrealized. The practical competitiveness of such an implementation is unknown.

Chapter 5

Conclusion

> A point in every direction is the same as no point at all.
>
> Harry Nilsson. "The Pointed Man", *The Point!* (1971).

For a nonsingular matrix $A \in \mathbb{Z}^{n \times n}$ together with a lifting modulus $X \in \mathbb{Z}_{>2}$ such that $X \perp \det A$, Chapter 2 describes an algorithm for "double-plus-one lifting" (Algorithm 4; Theorem 7) to compute a high-order residue $R \in \mathbb{Z}^{n \times n}$ such that $BA = I + RX^k$, together with a sparse inverse expansion

$$\Big(((B_0(I + R_0X) + M_0X^2)(I + R_1X^3) + M_1X^6) \cdots\Big)\Big((I + R_{k-2}X^{2^{k-1}-1}) + M_{k-2}X^{2^k-2}\Big)$$

for the matrix $B \equiv A^{-1} \bmod X^{2^k-1}$ in at most $O((\log n)n^\omega \mathsf{M}(\log n + \log ||A||))$ bit operations (Corollary 8). Additionally, this algorithm forms the basis of a method to deterministically certify $A \in \mathbb{Z}^{n \times n}$ as unimodular (Algorithm 5; Theorem 10).

Chapter 3 describes a heuristic algorithm for the determinant and Hermite normal form built upon the high-order residue algorithm of Chapter 2, fast nonsingular rational system solving, and a procedure to compute a triangular factor of the Hermite normal form from a projection $A^{-1}v$. No asymptotic complexity is given, although the output is certified to be correct and empirical tests show the running time grows as $O\tilde{}(n^3 \log n)$.

Details of an optimized implementation of the preceding chapters' algorithms are provided in Chapter 4. Experiments compare our Hermite form implementation with the high-performance implementations provided by Sage and Magma across a range of inputs.

On aggregate, the implementation described here can safely be said to be the fastest known for integer Hermite normal form.

References

[1] J. Abbott, M. Bronstein, and T. Mulders. Fast deterministic computation of determinants of dense matrices. In S. Dooley, editor, *Proc. Int'l. Symp. on Symbolic and Algebraic Computation: ISSAC'99*, pages 197–204. ACM Press, New York, 1999.

[2] E. Bach and J. Shallit. *Algorithmic Number Theory*, volume 1 : Efficient Algorithms. MIT Press, 1996.

[3] D. Bini and V. Y. Pan. *Polynomial and Matrix Computations, Vol 1: Fundamental Algorithms*. Birkhauser, Boston, 1994.

[4] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28:135–151, 2001.

[5] W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. I. The user language. *Journal of Symbolic Computation*, 24(3-4):235–265, 1997.

[6] Z. Chen and A. Storjohann. A BLAS based C library for exact linear algebra on integer matrices. In M. Kauers, editor, *Proc. Int'l. Symp. on Symbolic and Algebraic Computation: ISSAC'05*, pages 92–99. ACM Press, New York, 2005.

[7] H. Cohen. *A Course in Computational Algebraic Number Theory*. Springer-Verlag, 1996.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2 edition, 2001.

[9] J. D. Dixon. Exact solution of linear equations using p-adic expansions. *Numer. Math.*, 40:137–141, 1982.

[10] W. Eberly, M. Giesbrecht, and G. Villard. Computing the determinant and Smith form of an integer matrix. In *Proc. 31st Ann. IEEE Symp. Foundations of Computer Science*, pages 675–685, 2000.

[11] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 2nd edition, 2003.

[12] M. Giesbrecht. *Nearly Optimal Algorithms for Canonical Matrix Forms*. PhD thesis, University of Toronto, 1993.

[13] P. Giorgi. *Arithmetic and algorithmic in exact linear algebra for the LinBox library*. PhD thesis, École normale supérieure de Lyon, LIP, Lyon, France, December 2004.

[14] K. Goto and R. A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, 2008.

[15] T. Granlund and the GMP development team. GNU MP: The GNU multiple precision arithmetic library, 2011. Edition 5.0.2. http://gmplib.org.

[16] S. Gupta, S. Sarkar, A. Storjohann, and J. Valeriote. Triangular x-basis decompositions and derandomization of linear algebra algorithms over $\mathsf{K}[x]$. *Journal of Symbolic Computation*, October 2011. Festschrift for the 60th Birthday of Joachim von zur Gathen. Accepted for publication.

[17] J. L. Hafner and K. S. McCurley. Asymptotically fast triangularization of matrices over rings. *SIAM Journal of Computing*, 20(6):1068–1083, December 1991.

[18] C. Hermite. Sur l'introduction des variables continues dans la théorie des nombres. *J. Reine Angew. Math.*, 41:191–216, 1851.

[19] G. Jäger and C. Wagner. Efficient parallelizations of Hermite and Smith normal form algorithms. *Parallel Comput.*, 35(6):345–357, 2009.

[20] E. Kaltofen. On computing determinants of matrices without divisions. In P. S. Wang, editor, *Proc. Int'l. Symp. on Symbolic and Algebraic Computation: ISSAC'92*, pages 342–349. ACM Press, New York, 1992.

[21] E. Kaltofen and G. Villard. Computing the sign or the value of the determinant of an integer matrix, a complexity survey. *J. Computational Applied Math.*, 162(1):133–146, January 2004. Special issue: Proceedings of the International Conference on Linear Algebra and Arithmetic 2001, held in Rabat, Morocco, 28–31 May 2001, S. El Hajji, N. Revol, P. Van Dooren (guest eds.).

[22] E. Kaltofen and G. Villard. On the complexity of computing determinants. *Computational Complexity*, 13(3–4):91–130, 2004.

[23] R. Kannan and A. Bachem. Polynomial algorithms for computing the Smith and Hermite normal forms of and integer matrix. *SIAM Journal of Computing*, 8(4):499–507, November 1979.

[24] D. Micciancio and B. Warinschi. A linear space algorithm for computing the Hermite Normal Form. In B. Mourrain, editor, *Proc. Int'l. Symp. on Symbolic and Algebraic Computation: ISSAC'01*, pages 231–236. ACM, 2001.

[25] T. Mulders and A. Storjohann. Diophantine linear system solving. In S. Dooley, editor, *Proc. Int'l. Symp. on Symbolic and Algebraic Computation: ISSAC'99*, pages 281–288. ACM Press, New York, 1999.

[26] M. Newman. *Integral Matrices*. Academic Press, 1972.

[27] V. Y. Pan. Computing the determinant and the charactersitic polynomial of a matrix via solving linear systems of equations. *Inf. Proc. Letters*, 28:71–75, 1988.

[28] C. Pauderis and A. Storjohann. Deterministic unimodularity certification. In J. van der Hoeven and M. van Hoeij, editors, *Proc. Int'l. Symp. on Symbolic and Algebraic Computation: ISSAC'12*, pages 281–288. ACM Press, New York, 2012.

[29] C. Pauderis and A. Storjohann. Computing the invariant structure of integer matrices: fast algorithms into practice. In M. Kauers, editor, *Proc. Int'l. Symp. on Symbolic and Algebraic Computation: ISSAC'13*. ACM Press, New York, 2013.

[30] C. Pernet and W. Stein. Fast computation of Hermite normal forms of integer matrices. *Journal of Number Theory*, 130(7), 2010.

[31] K.C. Posch and R. Posch. Base extension using a convolution sum in residue number systems. *Computing*, 50(2):93–104, 1993.

[32] P. P. Shenoy and R. Kumaresan. Fast base extension using a redundant modulus in RNS. *IEEE Trans. Comput.*, 38(2):292–297, February 1989.

[33] A. Steel. Hermite normal form timings page. http://magma.maths.usyd.edu.au/users/allan/mat/hermite.html.

[34] W. A. Stein. Benchmarketing: Modular Hermite normal form. http://sagemath.blogspot.ca/2008/02/benchmarketing-modular-hermite-normal.html.

[35] W. A. Stein et al. *Sage Mathematics Software (Version 5.5.0)*. The Sage Development Team, 2012. http://www.sagemath.org.

[36] A. Storjohann. Computing Hermite and Smith normal forms of triangular integer matrices. *Linear Algebra and its Applications*, 282:25–45, 1998.

[37] A. Storjohann. *Algorithms for Matrix Canonical Forms*. PhD thesis, Swiss Federal Institute of Technology, ETH–Zurich, 2000.

[38] A. Storjohann. The shifted number system for fast linear algebra on integer matrices. *Journal of Complexity*, 21(4):609–650, 2005. Festschrift for the 70th Birthday of Arnold Schönhage.

[39] A. Storjohann and G. Labahn. Asymptotically fast computation of Hermite normal forms of integer matrices. In Y. N. Lakshman, editor, *Proc. Int'l. Symp. on Symbolic and Algebraic Computation: ISSAC'96*, pages 259–266. ACM Press, New York, 1996.

[40] Z. Wan. *Computing the Smith Forms of Integer Matrices and Solving Related Problems*. PhD thesis, University of Delaware, 2005.

[41] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the atlas project. *Parallel Computing*, 27(1-2), 2001.