Fast constant-time gcd computation and modular inversion

Daniel J. Bernstein[1,2] and Bo-Yin Yang[3]

[1] Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60607–7045, USA
[2] Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany
djb@cr.yp.to
[3] Institute of Information Science and Research Center of Information Technology and Innovation, Academia Sinica, 128 Section 2 Academia Road, Taipei 115-29, Taiwan
by@crypto.tw

Abstract. This paper introduces streamlined constant-time variants of Euclid's algorithm, both for polynomial inputs and for integer inputs. As concrete applications, this paper saves time in (1) modular inversion for Curve25519, which was previously believed to be handled much more efficiently by Fermat's method, and (2) key generation for the `ntruhrss701` and `sntrup4591761` lattice-based cryptosystems.

Keywords: Euclid's algorithm · greatest common divisor · gcd · modular reciprocal · modular inversion · constant-time computations · branchless algorithms · algorithm design · NTRU · Curve25519

1 Introduction

There is a vast literature on variants and applications of Euclid's gcd algorithm. A textbook extension of the algorithm computes modular reciprocals. Stopping the algorithm halfway produces a "half-gcd", writing one input modulo another as a ratio of two half-size outputs, an example of two-dimensional lattice-basis reduction. Well-known speedups include Lehmer's use of lower precision [44]; "binary" algorithms such as Stein's gcd algorithm [63] and Kaliski's "almost inverse" algorithm [40]; and the subquadratic algorithms of Knuth [41] and Schönhage [60], which combine Lehmer's idea with subquadratic integer multiplication. All of these algorithms have been adapted from the case of integer inputs to the (simpler) case of polynomial inputs, starting with Stevin's 16th-century algorithm [64, page 241 of original, page 123 of cited PDF] to compute the gcd of two polynomials. The Berlekamp–Massey algorithm (see [9] and [50]), one of the most important tools in decoding error-correcting codes, was later recognized as being equivalent to a special case of a Euclid–Stevin half-gcd computation, with the input and output coefficients in reverse order, and with one input being a power of x; see [32], [38], and [51]. There are many more examples.

Author list in alphabetical order; see https://www.ams.org/profession/leaders/culture/CultureStatement04.pdf. This work was supported by the U.S. National Science Foundation under grant 1314919, by the Cisco University Research Program, by the Netherlands Organisation for Scientific Research (NWO) under grant 639.073.005, and by DFG Cluster of Excellence 2092 "CASA: Cyber Security in the Age of Large-Scale Adversaries". This work also was supported by Taiwan Ministry of Science and Technology (MoST) grant MOST105-2221-E-001-014-MY3 and Academia Sinica Investigator Award AS-IA-104-M01. "Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation" (or other funding agencies). Permanent ID of this document: `c130922fff0455e43cc7c5ca180787781b409f63`. Date: 2019.03.05.

However, in cryptography, these algorithms are dangerous. The central problem is that these algorithms have conditional branches that depend on the inputs. Often these inputs are secret, and the branches leak information to the attacker through cache timing, branch timing, etc. For example, Aldaya–García–Tapia–Brumley [7] recently presented a single-trace cache-timing attack successfully extracting RSA keys from OpenSSL's implementation of Stein's binary gcd algorithm.

A common defense against these attacks[1] is to compute modular reciprocals in a completely different way, namely via Fermat's little theorem: if f is a nonzero element of a finite field $\mathbf{F}_q$ then $1/f = f^{q-2}$. For example, Bernstein's Curve25519 paper [10] computes a ratio modulo $p = 2^{255} - 19$ using "a straightforward sequence of 254 squarings and 11 multiplications" and says that this is "about 7% of the Curve25519 computation". Somewhat more multiplications are required for "random" primes, as in the new CSIDH [28] post-quantum system, but the bottom line is that a b-bit Fermat-style inversion is not much more expensive than b squarings, while a b-bit elliptic-curve variable-base-point scalar multiplication is an order of magnitude more expensive.

However, the bigger picture is that simply saying "use Fermat for inversion and stop worrying about the speed of Euclid's algorithm" is not satisfactory:

- There are many cryptographic computations where inversion is a much larger fraction of the total time. For example, typical strategies for ECC signature generation (see, e.g., [16, Section 4]) involve only about 1/4 as many elliptic-curve operations, making the cost of inversion[2] much more noticeable. As another example, polynomial inversion is the primary bottleneck in key generation[3] in the NTRU lattice-based cryptosystem.

- There are many cryptographic computations that rely on Euclid's algorithm for operations that Fermat's method cannot handle. For example, decryption in the McEliece code-based cryptosystem (see, e.g., [13]) relies critically on a half-gcd computation. This cryptosystem has large public keys but has attracted interest[4] for its very strong security track record. Many newer proposals with smaller public keys also rely on half-gcd computations.[5]

These issues have prompted some work on variants of Euclid's algorithm protected against timing attacks. For example, [13] uses a constant-time version of the Berlekamp–Massey algorithm inside the McEliece cryptosystem; see also [36, Algorithm 10] and [30]. As another example, Bos [21] developed a constant-time version of Kaliski's algorithm, and reported in [21, Table 1] that this took 486000 ARM Cortex-A8 cycles for inversion modulo a 254-bit prime. For comparison, Bernstein and Schwabe [17] had reported 527102 ARM Cortex-A8 cycles for an entire Curve25519 scalar multiplication, including Fermat inversion an order of magnitude faster than the Euclid inversion in [21].

Euclid's algorithm becomes more competitive for "random" moduli, since these moduli make the modular reductions in Fermat's method more expensive.[6] It is also easy to see

[1] There is also a tremendous amount of work aimed at defending against power attacks, electromagnetic attacks, and, more broadly, attackers with physical sensors close to the computation being carried out. For example, one typically tries to protect modular inversion by multiplying by r before and after each inversion, where r is a random invertible quantity. This paper focuses on protecting deterministic algorithms against timing attacks.

[2] This refers to inversion in $\mathbf{F}_p$ when the elliptic curve is defined over $\mathbf{F}_p$, to convert an elliptic-curve point from projective coordinates to affine coordinates. Applications that use ECDSA, rather than Schnorr-style signature systems such as Ed25519, also need inversions modulo the group order.

[3] The standard argument that key-generation performance is important is that, for forward secrecy, TLS generates a public key and uses the key just once. For several counterarguments see [14, Appendix T.2].

[4] For example, two of the round-2 submissions to the NIST post-quantum competition, Classic McEliece [12] and NTS-KEM [6], are based directly on the McEliece cryptosystem.

[5] See, e.g., the HQC [4] and LAC [45] round-2 submissions to the NIST competition.

[6] Standard estimates are that squaring modulo a random prime is 2 or 3 times slower than squaring

that Euclid's algorithm beats Fermat's method for all large enough primes: for n-bit primes, Euclid's algorithm uses $\Theta(n)$ simple linear-time operations such as big-integer subtractions, while Fermat's method uses $\Theta(n)$ big-integer multiplications. The Euclid-vs.-Fermat cutoff depends on the cost ratio between big-integer multiplications and big-integer subtractions, so one expects the cutoff to be larger on CPUs with large hardware multipliers, but many input sizes of interest in cryptography are clearly beyond the cutoff. Both [39, Algorithm 10] and [14, Appendix T.2] use constant-time variants of the Euclid–Stevin algorithm for polynomial inversions in NTRU key generation; each polynomial here has nearly 10000 bits in about 700 coefficients.

1.1. Contributions of this paper. This paper introduces simple fast constant-time "division steps". When division steps are iterated a constant number of times, they reveal the gcd of the inputs, the modular reciprocal when the inputs are coprime, etc. Division steps work the same way for integer inputs and for polynomial inputs, and are suitable for all applications of Euclid's algorithm that we have investigated. As an illustration, we display in Figure 1.2 an integer gcd algorithm using division steps.

This paper uses two case studies to show that these division steps are much faster than previous constant-time variants of Euclid's algorithm. One case study (Section 12) is an integer-modular-inversion problem selected to be extremely favorable to Fermat's method:

- The platforms are the popular Intel Haswell, Skylake, and Kaby Lake CPU cores. Each of these CPU cores has a large hardware area devoted to fast multiplications.
- The modulus is the Curve25519 prime $2^{255} - 19$. Sparse primes allow very fast reductions, and the performance of multiplication modulo this particular prime has been studied in detail on many platforms.

The latest speed records for inversion modulo $2^{255} - 19$ on the platforms mentioned above are from the recent Nath–Sarkar paper "Efficient inversion in (pseudo-)Mersenne prime order fields" [54], which takes 11854 cycles, 9301 cycles, and 8971 cycles on Haswell, Skylake, and Kaby Lake respectively. We achieve slightly better inversion speeds on each of these platforms: 10050 cycles, 8778 cycles, and 8543 cycles. We emphasize the Fermat-friendly nature of this case study. It is safe to predict that our algorithm will produce much larger speedups compared to Fermat's method on CPUs with smaller multipliers, on FPGAs, and on ASICs.[7] For example, on an ARM Cortex-A7 (only a 32-bit multiplier), our inversion modulo $2^{255} - 19$ takes 35277 cycles, while the best prior result (Fujii and Aranha [34], using Fermat) was 62648 cycles.

The other case study (Section 7) is key generation in the `ntruhrss701` cryptosystem from the Hülsing–Rijneveld–Schanck–Schwabe paper "High-speed key encapsulation from NTRU" [39] at CHES 2017. We again take an Intel core for comparability, specifically Haswell, since [39] selected Haswell. That paper used 150000 Haswell cycles to invert a polynomial modulo $(x^{701} - 1)/(x - 1)$ with coefficients modulo 3. We reduce the cost of this inversion to just 90000 Haswell cycles. We also speed up key generation in the `sntrup4591761` [14] cryptosystem.

1.3. Comparison to previous constant-time algorithms. Earlier constant-time variants of Euclid's algorithm and the Euclid–Stevin algorithm are like our algorithm in that they consist of a prespecified number of constant-time steps. Each step of the previous algorithms might seem rather simple at first glance. However, as illustrated by our new

modulo a special prime. Combining these estimates with the ARM Cortex-A8 speeds from [17] suggests that Fermat inversion for a random 256-bit prime should take roughly 100000 Cortex-A8 cycles, still much faster than the Euclid inversion from [21]. It is claimed in [21, Table 2] that Fermat inversion takes 584000 Cortex-A8 cycles; presumably this Fermat software was not properly optimized.

[7] We also expect spinoffs in quantum computation. See, e.g., the recent constant-time version of Kaliski's algorithm by Roetteler–Naehrig–Svore–Lauter [57, Section 3.4], in the context of Shor's algorithm to compute elliptic-curve discrete logarithms.

```
def shortgcd2(f,g):
  delta,f,g = 1,ZZ(f),ZZ(g)
  assert f&1
  m = 4+3*max(f.nbits(),g.nbits())
  for loop in range(m):
    if delta>0 and g&1: delta,f,g = -delta,g,-f
    delta,g = 1+delta,(g+(g&1)*f)//2
  return abs(f)
```

Figure 1.2: Example of a gcd algorithm for integers using this paper's division steps. The f input is assumed to be odd. Each iteration of the main loop replaces (δ, f, g) with $\operatorname{divstep}(\delta, f, g)$. Correctness follows from Theorem 11.2. The algorithm is expressed in the Sage [58] computer-algebra system. The algorithm is not constant-time as shown but can be made constant-time with low overhead; see Section 5.2.

speed records, the details matter. Our division steps are particularly simple, allowing very efficient computations.

Each step of the constant-time algorithms of Bos [21] and Roetteler–Naehrig–Svore–Lauter [57, Section 3.4], like each step in the algorithms of Stein and Kaliski, consists of a limited number of additions and subtractions, followed by a division by 2. The decisions of which operations to perform are based on (1) the bottom bits of the integers and (2) which integer is larger—a comparison that sometimes needs to inspect many bits of the integers. A constant-time comparison inspects all bits.

The time for this comparison might not seem problematic in context: subtraction also inspects all bits and produces the result of the comparison as a side effect. But this requires a very wide data flow in the algorithm. One cannot decide what needs to be be done in a step without inspecting all bits produced by the previous step. For our algorithm, the bottom t bits of the inputs (or the bottom t coefficients in the polynomial case) decide the linear combinations used in the next t division steps, so the data flow is much narrower.

For the polynomial case, size comparison is simply a matter of comparing polynomial degrees, but there are still many other details that affect performance, as illustrated by our 1.7× speedup for the `ntruhrss701` polynomial-inversion problem mentioned above. See Section 7 for a detailed comparison of our algorithm in this case to the algorithm in [39]. Important issues include the number of control variables manipulated in the inner loop, the complexity of the arithmetic operations being carried out, and the way that the output is scaled.

The closest previous algorithms that we have found in the literature are algorithms for "systolic" hardware developed in the 1980s by Bojanczyk, Brent, and Kung. See [24] for integer gcd, [19] for integer inversion, and [23] for the polynomial case. These algorithms have predictable output scaling; a single control variable δ (optionally represented in unary) beyond the loop counter; and the same basic feature that several bottom bits decide linear combinations used for several steps. Our algorithms are nevertheless simpler and faster, for a combination of reasons. First, our division steps have fewer case distinctions than the steps in [24] (and [19]). Second, our data flow can be decomposed into a "conditional swap" followed by an "elimination" (see Section 3), whereas the data flow in [23] requires a more complicated routing of inputs. Third, surprisingly, we use fewer iterations than [24]. Fourth, for us t bits determine linear combinations for exactly t steps, whereas this is not exactly true in [24]. Fifth, we gain considerable speed in, e.g., Curve25519 inversion by jumping through division steps.

1.4. Comparison to previous subquadratic algorithms. An easy consequence of our data flow is a constant-time algorithm where the number of operations is subquadratic

in the input size—asymptotically $n(\log n)^{2+o(1)}$. This algorithm has important advantages over earlier subquadratic gcd algorithms, as we now explain.

The starting point for subquadratic gcd algorithms is the following idea from Lehmer [44]: The quotients at the beginning of gcd computation depend only on the most significant bits of the inputs. After computing the corresponding 2×2 transition matrix, one can retroactively multiply this matrix by the remaining bits of the inputs, and then continue with the new, hopefully smaller, inputs.

A closer look shows that it is not easy to formulate a precise statement about the amount of progress that one is guaranteed to make by inspecting j most significant bits of the inputs. We return to this difficulty below. Assume for the moment that j bits of the inputs determine roughly j bits of initial information about the quotients in Euclid's algorithm.

Lehmer's paper predated subquadratic multiplication, but is well known to be helpful even with quadratic multiplication. For example, consider the following procedure starting with 40-bit integers a and b: add a to b, then add the new b to a, and so on back and forth for a total of 30 additions. The results fit into words on a typical 64-bit CPU, and this procedure might seem to be efficiently using the CPU's arithmetic instructions. However, it is generally faster to directly compute $832040a + 514229b$ and $1346269a + 832040b$, using the CPU's fast multiplication circuits.

Faster algorithms for multiplication make Lehmer's approach more effective. Knuth [41], using Lehmer's idea recursively on top of FFT-based integer multiplication, achieved cost $n(\log n)^{5+o(1)}$ for integer gcd. Schönhage [60] improved 5 to 2. Subsequent papers such as [53], [22], and [66] adapted the ideas to the polynomial case.

The subquadratic algorithms appearing in [41], [60], [22, Algorithm EMGCD], [66, Algorithm SCH], [62, Algorithm Half-GB-gcd], [52, Algorithms HGCD-Q and HGCD-B and SGCD and HGCD-D], [25, Algorithm 3.1], etc. all have the following structure. There is an initial recursive call that computes a 2×2 transition matrix; there is a computation of reduced inputs, using a fast multiplication subroutine; there is a call to a fast division subroutine, producing a possibly large quotient and remainder; and then there is more recursion.

We emphasize the role of the division subroutine in each of these algorithms. If the recursive computation is expressed as a tree then each node in the tree calls the left subtree recursively, multiplies by a transition matrix, calls a division subroutine, and calls the right subtree recursively. The results of the left subtree are described by some number of quotients in Euclid's algorithm, so variations in the sizes of quotients produce irregularities in the amount of information computed by the recursions. These irregularities can be quite severe: multiplying by the transition matrix does not make any progress when the inputs have sufficiently different sizes. The call to a division subroutine is a bandage over these irregularities, guaranteeing significant progress in all cases.

Our subquadratic algorithm has a simpler structure. We guarantee that a recursive call to a size-j subtree always jumps through exactly j division steps. Each node in the computation tree involves multiplications and recursive calls, but the large variable-time division subroutine has disappeared, and the underlying irregularities have also disappeared. One can think of each Euclid-style division as being decomposed into a series of our simpler constant-time division steps, but our recursion does not care where each Euclid-style division begins and ends. See Section 12 for a case study of the speeds that we achieve.

A prerequisite for this regular structure is that the linear combinations in j of our division steps are determined by the bottom j bits of the integer inputs[8] without regard to top bits. This prerequisite was almost achieved by the Brent–Kung "systolic" gcd algorithm [24] mentioned above, but Brent and Kung did not observe that the data flow

[8]For polynomials one can work from the bottom or from the top, since there are no carries. We choose to work from bottom coefficients in the polynomial case, to emphasize the similarities between the polynomial case and the integer case.

allows a subquadratic algorithm. The Stehlé–Zimmermann subquadratic "binary recursive gcd" algorithm [62] also emphasizes that it works with bottom bits, but it has the same irregularities as earlier subquadratic algorithms, and it again needs a large variable-time division subroutine.

Often the literature considers the "normal" case of polynomial gcd computation. This is, by definition, the case that each Euclid–Stevin quotient has degree 1. The conventional recursion then returns a predictable amount of information, allowing a more regular algorithm, such as [53, Algorithm PGCD].[9] Our algorithm achieves the same regularity for the general case.

Another previous algorithm that achieves the same regularity for a different special case of polynomial gcd computation is Thomé's subquadratic variant [68, Program 4.1] of the Berlekamp–Massey algorithm. Our algorithm can be viewed as extending this regularity to arbitrary polynomial gcd computations, and, beyond this, to integer gcd computations.

2 Organization of the paper

We start with the polynomial case. Section 3 defines division steps. Section 5, relying on theorems in Section 4, states our main algorithm to compute c coefficients of the nth iterate of divstep. This takes $n(c+n)$ simple operations. We also explain how "jumps" reduce the cost for large n to $(c+n)(\log cn)^{2+o(1)}$ operations. All of these algorithms take constant time, i.e., time independent of the input coefficients for any particular (n, c).

There is a long tradition in number theory of defining Euclid's algorithm, continued fractions, etc. for inputs in a "complete" field, such as the field of real numbers or the field of power series. We define division steps for power series, and state our main algorithm for power series. Division steps produce polynomial outputs from polynomial inputs, so the reader can restrict attention to polynomials if desired, but there are advantages of following the tradition, as we now explain.

Requiring algorithms to work for general power series means that the algorithms can only inspect a limited number of leading coefficients of each input, without regard to the degree of inputs that happen to be polynomials. This restriction simplifies algorithm design, and helped us design our main algorithm.

Going beyond polynomials also makes it easy to describe further algorithmic options, such as iterating divstep on inputs $(1, g/f)$ instead of (f, g). Restricting to polynomial inputs would require g/f to be replaced with a nearby polynomial; the limited precision of the inputs would affect the table of divstep iterates, rather than merely being an option in the algorithms.

Section 6, relying on theorems in Appendix A, applies our main algorithm to gcd computation and modular inversion. Here the inputs are polynomials. A constant fraction of the power-series coefficients used in our main algorithm are guaranteed to be 0 in this case, and one can save some time by skipping computations involving those coefficients. However, it is still helpful to factor the algorithm-design process into (1) designing the fast power-series algorithm and then (2) eliminating unused values for special cases.

Section 7 is a case study of the software speed of modular inversion of polynomials as part of key generation in the NTRU cryptosystem.

The remaining sections of the paper consider the integer case. Section 8 defines division steps for integers. Section 10, relying on theorems in Section 9, states our main algorithm to compute c bits of the nth iterate of divstep. Section 11, relying on theorems in Appendix G, applies our main algorithm to gcd computation and modular inversion. Section 12 is a case

[9] The analysis in [53, Section VI] considers only normal inputs (and power-of-2 sizes). The algorithm works only for normal inputs. The performance of division in the non-normal case was discussed in [53, Section VIII] but incorrectly described as solely an issue for the base case of the recursion.

study of the software speed of inversion of integers modulo primes used in elliptic-curve cryptography.

2.1. General notation. We write $M_2(R)$ for the ring of 2×2 matrices over a ring R. For example, $M_2(\mathbf{R})$ is the ring of 2×2 matrices over the field $\mathbf{R}$ of real numbers.

2.2. Notation for the polynomial case. Fix a field k. The formal-power-series ring $k[[x]]$ is the set of infinite sequences $(f_0, f_1, \dots)$ with each $f_i \in k$. The ring operations are

- 0: $(0, 0, 0, \dots)$.
- 1: $(1, 0, 0, \dots)$.
- $+$: $(f_0, f_1, \dots), (g_0, g_1, \dots) \mapsto (f_0 + g_0, f_1 + g_1, \dots)$.
- $-$: $(f_0, f_1, \dots), (g_0, g_1, \dots) \mapsto (f_0 - g_0, f_1 - g_1, \dots)$.
- $\cdot$: $(f_0, f_1, f_2, \dots), (g_0, g_1, g_2, \dots) \mapsto (f_0 g_0, f_0 g_1 + f_1 g_0, f_0 g_2 + f_1 g_1 + f_2 g_0, \dots)$.

The element $(0, 1, 0, \dots)$ is written "x", and the entry f_i in $f = (f_0, f_1, \dots)$ is called the "coefficient of x^i in f". As in the Sage [58] computer-algebra system, we write $f[i]$ for the coefficient of x^i in f. The "constant coefficient" $f[0]$ has a more standard notation $f(0)$, and we use the $f(0)$ notation when we are not also inspecting other coefficients.

The unit group of $k[[x]]$ is written $k[[x]]^*$. This is the same as the set of $f \in k[[x]]$ such that $f(0) \neq 0$.

The polynomial ring $k[x]$ is the subring of sequences $(f_0, f_1, \dots)$ that have only finitely many nonzero coefficients. We write $\deg f$ for the degree of $f \in k[x]$; this means the maximum i such that $f[i] \neq 0$, or $-\infty$ if $f = 0$. We also define $f[-\infty] = 0$, so $f[\deg f]$ is always defined. Beware that the literature sometimes instead defines $\deg 0 = -1$, and in particular Sage defines $\deg 0 = -1$.

We define $f \bmod x^n$ as $f[0] + f[1]x + \cdots + f[n-1]x^{n-1}$ for any $f \in k[[x]]$. We define $f \bmod g$ for any $f, g \in k[x]$ with $g \neq 0$ as the unique polynomial r such that $\deg r < \deg g$ and $f - r = gq$ for some $q \in k[x]$.

The ring $k[[x]]$ is a domain. The formal-Laurent-series field $k((x))$ is, by definition, the field of fractions of $k[[x]]$. Each element of $k((x))$ can be written (not uniquely) as f/x^i for some $f \in k[[x]]$ and some $i \in \{0, 1, 2, \dots\}$. The subring $k[1/x]$ is the smallest subring of $k((x))$ containing both k and $1/x$.

2.3. Notation for the integer case. The set of infinite sequences $(f_0, f_1, \dots)$ with each $f_i \in \{0, 1\}$ forms a ring under the following operations:

- 0: $(0, 0, 0, \dots)$.
- 1: $(1, 0, 0, \dots)$.
- $+$: $(f_0, f_1, \dots), (g_0, g_1, \dots) \mapsto$ the unique $(h_0, h_1, \dots)$ such that, for every $n \geq 0$: $h_0 + 2h_1 + 4h_2 + \cdots + 2^{n-1}h_{n-1} \equiv (f_0 + 2f_1 + 4f_2 + \cdots + 2^{n-1}f_{n-1}) + (g_0 + 2g_1 + 4g_2 + \cdots + 2^{n-1}g_{n-1}) \pmod{2^n}$.
- $-$: same with $(f_0 + 2f_1 + 4f_2 + \cdots + 2^{n-1}f_{n-1}) - (g_0 + 2g_1 + 4g_2 + \cdots + 2^{n-1}g_{n-1})$.
- $\cdot$: same with $(f_0 + 2f_1 + 4f_2 + \cdots + 2^{n-1}f_{n-1}) \cdot (g_0 + 2g_1 + 4g_2 + \cdots + 2^{n-1}g_{n-1})$.

We follow the tradition among number theorists of using the notation $\mathbf{Z}_2$ for this ring, and not for the finite field $\mathbf{F}_2 = \mathbf{Z}/2$. This ring $\mathbf{Z}_2$ has characteristic 0, so $\mathbf{Z}$ can be viewed as a subset of $\mathbf{Z}_2$.

One can think of $(f_0, f_1, \dots) \in \mathbf{Z}_2$ as an infinite series $f_0 + 2f_1 + \cdots$. The difference between $\mathbf{Z}_2$ and the power-series ring $\mathbf{F}_2[[x]]$ is that $\mathbf{Z}_2$ has carries: for example, adding $(1, 0, \dots)$ to $(1, 0, \dots)$ in $\mathbf{Z}_2$ produces $(0, 1, \dots)$.

We define $f \bmod 2^n$ as $f_0 + 2f_1 + \cdots + 2^{n-1}f_{n-1}$ for any $f = (f_0, f_1, \dots) \in \mathbf{Z}_2$.

The unit group of $\mathbf{Z}_2$ is written $\mathbf{Z}_2^*$. This is the same as the set of odd $f \in \mathbf{Z}_2$, i.e., the set of $f \in \mathbf{Z}_2$ such that $f \bmod 2 \neq 0$.

If $f \in \mathbf{Z}_2$ and $f \neq 0$ then $\operatorname{ord}_2 f$ means the largest integer $e \geq 0$ such that $f \in 2^e\mathbf{Z}_2$; equivalently, the unique integer $e \geq 0$ such that $f \in 2^e\mathbf{Z}_2^*$.

The ring $\mathbf{Z}_2$ is a domain. The field $\mathbf{Q}_2$ is, by definition, the field of fractions of $\mathbf{Z}_2$. Each element of $\mathbf{Q}_2$ can be written (not uniquely) as $f/2^i$ for some $f \in \mathbf{Z}_2$ and some $i \in \{0, 1, 2, \dots\}$. The subring $\mathbf{Z}[1/2]$ is the smallest subring of $\mathbf{Q}_2$ containing both $\mathbf{Z}$ and $1/2$.

3 Definition of x-adic division steps

Define $\operatorname{divstep} : \mathbf{Z} \times k[[x]]^* \times k[[x]] \to \mathbf{Z} \times k[[x]]^* \times k[[x]]$ as follows:

$$\operatorname{divstep}(\delta, f, g) = \begin{cases} (1-\delta, g, (g(0)f - f(0)g)/x) & \text{if } \delta > 0 \text{ and } g(0) \neq 0, \\ (1+\delta, f, (f(0)g - g(0)f)/x) & \text{otherwise.} \end{cases}$$

Note that $f(0)g - g(0)f$ is divisible by x in $k[[x]]$. The name "division step" is justified in Theorem C.1, which shows that dividing a degree-d_0 polynomial by a degree-d_1 polynomial with $0 \leq d_1 < d_0$ can be viewed as computing $\operatorname{divstep}^{2d_0-2d_1}$.

3.1. Transition matrices. Write $(\delta_1, f_1, g_1) = \operatorname{divstep}(\delta, f, g)$. Then

$$\begin{pmatrix} f_1 \\ g_1 \end{pmatrix} = \mathcal{T}(\delta, f, g) \begin{pmatrix} f \\ g \end{pmatrix} \qquad \text{and} \qquad \begin{pmatrix} 1 \\ \delta_1 \end{pmatrix} = \mathcal{S}(\delta, f, g) \begin{pmatrix} 1 \\ \delta \end{pmatrix}$$

where $\mathcal{T} : \mathbf{Z} \times k[[x]]^* \times k[[x]] \to M_2(k[1/x])$ is defined by

$$\mathcal{T}(\delta, f, g) = \begin{cases} \begin{pmatrix} 0 & 1 \\ \frac{g(0)}{x} & \frac{-f(0)}{x} \end{pmatrix} & \text{if } \delta > 0 \text{ and } g(0) \neq 0, \\ \begin{pmatrix} 1 & 0 \\ \frac{-g(0)}{x} & \frac{f(0)}{x} \end{pmatrix} & \text{otherwise,} \end{cases}$$

and $\mathcal{S} : \mathbf{Z} \times k[[x]]^* \times k[[x]] \to M_2(\mathbf{Z})$ is defined by

$$\mathcal{S}(\delta, f, g) = \begin{cases} \begin{pmatrix} 1 & 0 \\ 1 & -1 \end{pmatrix} & \text{if } \delta > 0 \text{ and } g(0) \neq 0, \\ \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} & \text{otherwise.} \end{cases}$$

Note that both $\mathcal{T}(\delta, f, g)$ and $\mathcal{S}(\delta, f, g)$ are defined entirely by $(\delta, f(0), g(0))$; they do not depend on the remaining coefficients of f and g.

3.2. Decomposition. This divstep function is a composition of two simpler functions (and the transition matrices can similarly be decomposed):

- a **conditional swap** replacing (δ, f, g) with $(-\delta, g, f)$ if $\delta > 0$ and $g(0) \neq 0$; followed by
- an **elimination** replacing (δ, f, g) with $(1+\delta, f, (f(0)g - g(0)f)/x)$.

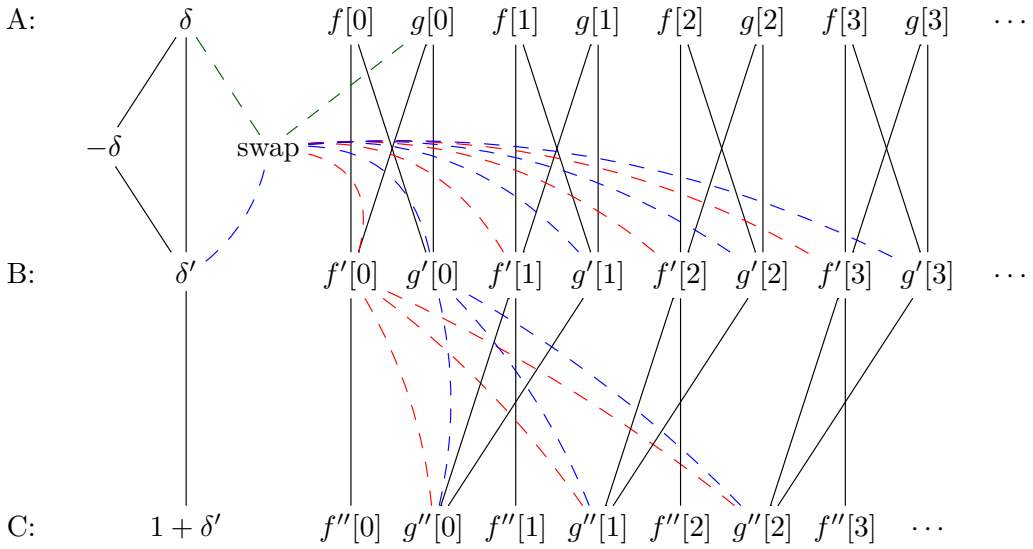

Figure 3.3: Data flow in an x-adic division step decomposed as a conditional swap (A to B) followed by an elimination (B to C). The swap bit is set if $\delta > 0$ and $g[0] \neq 0$. The g outputs are $f'[0]g'[1] - g'[0]f'[1]$, $f'[0]g'[2] - g'[0]f'[2]$, $f'[0]g'[3] - g'[0]f'[3]$, etc.

See Figure 3.3.

This decomposition is our motivation for defining divstep in the first case—the swapped case—to compute $(g(0)f - f(0)g)/x$. Using $(f(0)g - g(0)f)/x$ in both cases might seem simpler but would require the conditional swap to forward a conditional negation to the elimination.

One can also compose these functions in the opposite order. This is compatible with some of what we say later about iterating the composition. However, the corresponding transition matrices would depend on another coefficient of g. This would complicate our algorithm statements.

Another possibility, as in [23], is to keep f and g in place, using the sign of δ to decide whether to add a multiple of f into g or to add a multiple of g into f. One way to do this in constant time is to perform two multiplications and two additions. Another way is to perform conditional swaps before and after one addition and one multiplication. Our approach is more efficient.

3.4. Scaling. One can define a variant of divstep that computes $f - (f(0)/g(0))g$ in the first case (the swapped case) and $g - (g(0)/f(0))f$ in the second case.

This variant has two advantages. First, it multiplies only one series, rather than two, by a scalar in k. Second, multiplying both f and g by any $u \in k[[x]]^*$ has the same effect on the output, so this variant induces a function on fewer variables $(\delta, g/f)$: the ratio $\rho = g/f$ is replaced by $(1/\rho - 1/\rho(0))/x$ when $\delta > 0$ and $\rho(0) \neq 0$, and by $(\rho - \rho(0))/x$ otherwise, while δ is replaced by $1-\delta$ or $1+\delta$ respectively. This is the composition of, first, a conditional inversion that replaces (δ, ρ) with $(-\delta, 1/\rho)$ if $\delta > 0$ and $\rho(0) \neq 0$; second, an elimination that replaces ρ with $(\rho - \rho(0))/x$, while adding 1 to δ.

On the other hand, working projectively with f, g is generally more efficient than working with ρ. Working with $f(0)g - g(0)f$ rather than $g - (g(0)/f(0))f$ has the virtue of being a "fraction-free" computation: it involves only ring operations in k, not divisions. This makes the effect of scaling only slightly more difficult to track. Specifically, say divstep$(\delta, f, g) =$

Table 4.1: Iterates $(\delta_n, f_n, g_n) = \operatorname{divstep}^n(\delta, f, g)$ for $k = \mathbf{F}_7$, $\delta = 1$, $f = 2 + 7x + 1x^2 + 8x^3 + 2x^4 + 8x^5 + 1x^6 + 8x^7$, and $g = 3 + 1x + 4x^2 + 1x^3 + 5x^4 + 9x^5 + 2x^6$. Each power series is displayed as a row of coefficients of x^0, x^1, etc.

n	δ_n	f_n											g_n										
		x^0	x^1	x^2	x^3	x^4	x^5	x^6	x^7	x^8	x^9	...	x^0	x^1	x^2	x^3	x^4	x^5	x^6	x^7	x^8	x^9	...
0	1	2	0	1	1	2	1	1	1	0	0	...	3	1	4	1	5	2	2	0	0	0	...
1	0	3	1	4	1	5	2	2	0	0	0	...	5	2	1	3	6	6	3	0	0	0	...
2	1	3	1	4	1	5	2	2	0	0	0	...	1	4	4	0	1	6	0	0	0	0	...
3	0	1	4	4	0	1	6	0	0	0	0	...	3	6	1	2	5	2	0	0	0	0	...
4	1	1	4	4	0	1	6	0	0	0	0	...	1	3	2	2	5	0	0	0	0	0	...
5	0	1	3	2	2	5	0	0	0	0	0	...	1	2	5	3	6	0	0	0	0	0	...
6	1	1	3	2	2	5	0	0	0	0	0	...	6	3	1	1	0	0	0	0	0	0	...
7	0	6	3	1	1	0	0	0	0	0	0	...	1	4	4	2	0	0	0	0	0	0	...
8	1	6	3	1	1	0	0	0	0	0	0	...	0	2	4	0	0	0	0	0	0	0	...
9	2	6	3	1	1	0	0	0	0	0	0	...	5	3	0	0	0	0	0	0	0	0	...
10	−1	5	3	0	0	0	0	0	0	0	0	...	4	5	5	0	0	0	0	0	0	0	...
11	0	5	3	0	0	0	0	0	0	0	0	...	6	4	0	0	0	0	0	0	0	0	...
12	1	5	3	0	0	0	0	0	0	0	0	...	2	0	0	0	0	0	0	0	0	0	...
13	0	2	0	0	0	0	0	0	0	0	0	...	6	0	0	0	0	0	0	0	0	0	...
14	1	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	...
15	2	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	...
16	3	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	...
17	4	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	...
18	5	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	...
19	6	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

(δ', f', g'). If $u \in k[[x]]$ with $u(0) = 1$ then $\operatorname{divstep}(\delta, uf, ug) = (\delta', uf', ug')$. If $u \in k^*$ then

$$\operatorname{divstep}(\delta, uf, g) = (\delta', f', ug') \text{ in the first (swapped) case;}$$
$$\operatorname{divstep}(\delta, uf, g) = (\delta', uf', ug') \text{ in the second case;}$$
$$\operatorname{divstep}(\delta, f, ug) = (\delta', uf', ug') \text{ in the first case;}$$
$$\operatorname{divstep}(\delta, f, ug) = (\delta', f', ug') \text{ in the second case;}$$

and $\operatorname{divstep}(\delta, uf, ug) = (\delta', uf', u^2g')$.

One can also scale $g(0)f - f(0)g$ by other nonzero constants. For example, for typical fields k one can quickly find "half-size" a, b such that $a/b = g(0)/f(0)$. Computing $af - bg$ may be more efficient than computing $g(0)f - f(0)g$.

We use the $g(0)/f(0)$ variant in our case study in Section 7 for the field $k = \mathbf{F}_3$. Dividing by $f(0)$ in this field is the same as multiplying by $f(0)$.

4 Iterates of x-adic division steps

Starting from $(\delta, f, g) \in \mathbf{Z} \times k[[x]]^* \times k[[x]]$, define $(\delta_n, f_n, g_n) = \operatorname{divstep}^n(\delta, f, g) \in \mathbf{Z} \times k[[x]]^* \times k[[x]]$ for each $n \geq 0$. Also define $\mathcal{T}_n = \mathcal{T}(\delta_n, f_n, g_n) \in M_2(k[1/x])$ and $\mathcal{S}_n = \mathcal{S}(\delta_n, f_n, g_n) \in M_2(\mathbf{Z})$ for each $n \geq 0$. Our main algorithm relies on various simple mathematical properties of these objects; this section states those properties. Table 4.1 shows all δ_n, f_n, g_n for one example of (δ, f, g).

Theorem 4.2. $\begin{pmatrix} f_n \\ g_n \end{pmatrix} = \mathcal{T}_{n-1} \cdots \mathcal{T}_m \begin{pmatrix} f_m \\ g_m \end{pmatrix}$ *and* $\begin{pmatrix} 1 \\ \delta_n \end{pmatrix} = \mathcal{S}_{n-1} \cdots \mathcal{S}_m \begin{pmatrix} 1 \\ \delta_m \end{pmatrix}$ *if* $n \geq m \geq 0$.

Proof. This follows from $\begin{pmatrix} f_{m+1} \\ g_{m+1} \end{pmatrix} = \mathcal{T}_m \begin{pmatrix} f_m \\ g_m \end{pmatrix}$ and $\begin{pmatrix} 1 \\ \delta_{m+1} \end{pmatrix} = \mathcal{S}_m \begin{pmatrix} 1 \\ \delta_m \end{pmatrix}$. □

Theorem 4.3. $\mathcal{T}_{n-1} \cdots \mathcal{T}_m \in \begin{pmatrix} k + \cdots + \frac{1}{x^{n-m-1}} k & k + \cdots + \frac{1}{x^{n-m-1}} k \\ \frac{1}{x} k + \cdots + \frac{1}{x^{n-m}} k & \frac{1}{x} k + \cdots + \frac{1}{x^{n-m}} k \end{pmatrix}$ *if* $n > m \geq 0$.

A product of $n - m$ of these $\mathcal{T}$ matrices can thus be stored using $4(n - m)$ coefficients from k. Beware that the theorem statement relies on $n > m$: for $n = m$ the product is the identity matrix.

Proof. If $n - m = 1$ then the product is $\mathcal{T}_m$, which is in $\begin{pmatrix} k & k \\ \frac{1}{x} k & \frac{1}{x} k \end{pmatrix}$ by definition. For $n - m > 1$, assume inductively that

$$\mathcal{T}_{n-1} \cdots \mathcal{T}_{m+1} \in \begin{pmatrix} k + \cdots + \frac{1}{x^{n-m-2}} k & k + \cdots + \frac{1}{x^{n-m-2}} k \\ \frac{1}{x} k + \cdots + \frac{1}{x^{n-m-1}} k & \frac{1}{x} k + \cdots + \frac{1}{x^{n-m-1}} k \end{pmatrix}.$$

Multiply on the right by $\mathcal{T}_m \in \begin{pmatrix} k & k \\ \frac{1}{x} k & \frac{1}{x} k \end{pmatrix}$. The top entries of the product are in $(k + \cdots + \frac{1}{x^{n-m-2}} k) + (\frac{1}{x} k + \cdots + \frac{1}{x^{n-m-1}} k) = k + \cdots + \frac{1}{x^{n-m-1}} k$ as claimed, and the bottom entries are in $(\frac{1}{x} k + \cdots + \frac{1}{x^{n-m-1}} k) + (\frac{1}{x^2} k + \cdots + \frac{1}{x^{n-m}} k) = \frac{1}{x} k + \cdots + \frac{1}{x^{n-m}} k$ as claimed. □

Theorem 4.4. $\mathcal{S}_{n-1} \cdots \mathcal{S}_m \in \begin{pmatrix} 1 & 0 \\ \{2 - (n-m), \ldots, n-m-2, n-m\} & \{1, -1\} \end{pmatrix}$ *if* $n > m \geq 0$.

In other words, the bottom-left entry is between $-(n-m)$ exclusive and $n-m$ inclusive, and has the same parity as $n - m$. There are exactly $2(n - m)$ possibilities for the product. Beware that the theorem statement again relies on $n > m$.

Proof. If $n - m = 1$ then $\{2 - (n - m), \ldots, n - m - 2, n - m\} = \{1\}$ and the product is $\mathcal{S}_m$, which is $\left(\begin{smallmatrix} 1 & 0 \\ 1 & \pm 1 \end{smallmatrix}\right)$ by definition.

For $n - m > 1$, assume inductively that

$$\mathcal{S}_{n-1} \cdots \mathcal{S}_{m+1} \in \begin{pmatrix} 1 & 0 \\ \{2 - (n-m-1), \ldots, n-m-3, n-m-1\} & \{1, -1\} \end{pmatrix},$$

and multiply on the right by $\mathcal{S}_m = \left(\begin{smallmatrix} 1 & 0 \\ 1 & \pm 1 \end{smallmatrix}\right)$. The top two entries of the product are again 1 and 0. The bottom-left entry is in $\{2 - (n - m - 1), \ldots, n - m - 3, n - m - 1\} + \{1, -1\} = \{2 - (n - m), \ldots, n - m - 2, n - m\}$. The bottom-right entry is again 1 or -1. □

The next theorem compares $\delta_m, f_m, g_m, \mathcal{T}_m, \mathcal{S}_m$ to $\delta'_m, f'_m, g'_m, \mathcal{T}'_m, \mathcal{S}'_m$ defined in the same way starting from $(\delta', f', g') \in \mathbf{Z} \times k[[x]]^* \times k[[x]]$.

Theorem 4.5. *Let m, t be nonnegative integers. Assume that $\delta'_m = \delta_m$; $f'_m \equiv f_m \pmod{x^t}$; and $g'_m \equiv g_m \pmod{x^t}$. Then, for each integer n with $m < n \leq m + t$: $\mathcal{T}'_{n-1} = \mathcal{T}_{n-1}$; $\mathcal{S}'_{n-1} = \mathcal{S}_{n-1}$; $\delta'_n = \delta_n$; $f'_n \equiv f_n \pmod{x^{t-(n-m-1)}}$; and $g'_n \equiv g_n \pmod{x^{t-(n-m)}}$.*

In other words, δ_m and the first t coefficients of f_m and g_m determine all of the following:

- the matrices $\mathcal{T}_m, \mathcal{T}_{m+1}, \ldots, \mathcal{T}_{m+t-1}$;
- the matrices $\mathcal{S}_m, \mathcal{S}_{m+1}, \ldots, \mathcal{S}_{m+t-1}$;
- $\delta_{m+1}, \ldots, \delta_{m+t}$;
- the first t coefficients of f_{m+1}, the first $t - 1$ coefficients of f_{m+2}, the first $t - 2$ coefficients of f_{m+3}, and so on through the first coefficient of f_{m+t};

- the first $t-1$ coefficients of g_{m+1}, the first $t-2$ coefficients of g_{m+2}, the first $t-3$ coefficients of g_{m+3}, and so on through the first coefficient of g_{m+t-1}.

Our main algorithm computes $(\delta_n, f_n \bmod x^{t-(n-m-1)}, g_n \bmod x^{t-(n-m)})$ for any selected $n \in \{m+1, \ldots, m+t\}$, along with the product of the matrices $\mathcal{T}_m, \ldots, \mathcal{T}_{n-1}$. See Section 5.

Proof. See Figure 3.3 for the intuition. Formally, fix m, t and induct on n. Observe that $(\delta'_{n-1}, f'_{n-1}(0), g'_{n-1}(0))$ equals $(\delta_{n-1}, f_{n-1}(0), g_{n-1}(0))$:

- If $n = m+1$: By assumption $f'_m \equiv f_m \pmod{x^t}$, and $n \le m+t$ so $t \ge 1$, so in particular $f'_m(0) = f_m(0)$. Similarly $g'_m(0) = g_m(0)$. By assumption $\delta'_m = \delta_m$.
- If $n > m+1$: $f'_{n-1} \equiv f_{n-1} \pmod{x^{t-(n-m-2)}}$ by the inductive hypothesis, and $n \le m+t$ so $t-(n-m-2) \ge 2$, so in particular $f'_{n-1}(0) = f_{n-1}(0)$; similarly $g'_{n-1} \equiv g_{n-1} \pmod{x^{t-(n-m-1)}}$ by the inductive hypothesis, and $t-(n-m-1) \ge 1$, so in particular $g'_{n-1}(0) = g_{n-1}(0)$; and $\delta'_{n-1} = \delta_{n-1}$ by the inductive hypothesis.

Now use the fact that $\mathcal{T}$ and $\mathcal{S}$ inspect only the first coefficient of each series to see that

$$\mathcal{T}'_{n-1} = \mathcal{T}(\delta'_{n-1}, f'_{n-1}, g'_{n-1}) = \mathcal{T}(\delta_{n-1}, f_{n-1}, g_{n-1}) = \mathcal{T}_{n-1},$$
$$\mathcal{S}'_{n-1} = \mathcal{S}(\delta'_{n-1}, f'_{n-1}, g'_{n-1}) = \mathcal{S}(\delta_{n-1}, f_{n-1}, g_{n-1}) = \mathcal{S}_{n-1}$$

as claimed.

Multiply $\begin{pmatrix} 1 \\ \delta'_{n-1} \end{pmatrix} = \begin{pmatrix} 1 \\ \delta_{n-1} \end{pmatrix}$ by $\mathcal{S}'_{n-1} = \mathcal{S}_{n-1}$ to see that $\delta'_n = \delta_n$ as claimed.

By the inductive hypothesis, $(\mathcal{T}'_m, \ldots, \mathcal{T}'_{n-2}) = (\mathcal{T}_m, \ldots, \mathcal{T}_{n-2})$, and $\mathcal{T}'_{n-1} = \mathcal{T}_{n-1}$, so $\mathcal{T}'_{n-1} \cdots \mathcal{T}'_m = \mathcal{T}_{n-1} \cdots \mathcal{T}_m$. Abbreviate $\mathcal{T}_{n-1} \cdots \mathcal{T}_m$ as P. Then $\begin{pmatrix} f'_n \\ g'_n \end{pmatrix} = P \begin{pmatrix} f'_m \\ g'_m \end{pmatrix}$ and $\begin{pmatrix} f_n \\ g_n \end{pmatrix} = P \begin{pmatrix} f_m \\ g_m \end{pmatrix}$ by Theorem 4.2, so $\begin{pmatrix} f'_n - f_n \\ g'_n - g_n \end{pmatrix} = P \begin{pmatrix} f'_m - f_m \\ g'_m - g_m \end{pmatrix}$.

By Theorem 4.3, $P \in \begin{pmatrix} k + \cdots + \frac{1}{x^{n-m-1}}k & k + \cdots + \frac{1}{x^{n-m-1}}k \\ \frac{1}{x}k + \cdots + \frac{1}{x^{n-m}}k & \frac{1}{x}k + \cdots + \frac{1}{x^{n-m}}k \end{pmatrix}$. By assumption, $f'_m - f_m$ and $g'_m - g_m$ are multiples of x^t. Multiply to see that $f'_n - f_n$ is a multiple of $x^t/x^{n-m-1} = x^{t-(n-m-1)}$ as claimed, and $g'_n - g_n$ is a multiple of $x^t/x^{n-m} = x^{t-(n-m)}$ as claimed. □

5 Fast computation of iterates of x-adic division steps

Our goal in this section is to compute $(\delta_n, f_n, g_n) = \text{divstep}^n(\delta, f, g)$. We are given the power series f, g to precision t, t respectively, and we compute f_n, g_n to precision $t-n+1, t-n$ respectively if $n \ge 1$; see Theorem 4.5. We also compute the n-step transition matrix $\mathcal{T}_{n-1} \cdots \mathcal{T}_0$.

We begin with an algorithm that simply computes one divstep iterate after another, multiplying by scalars in k and dividing by x exactly as in the divstep definition. We specify this algorithm in Figure 5.1 as a function `divstepsx` in the Sage [58] computer-algebra system. The quantities $u, v, q, r \in k[1/x]$ inside the algorithm keep track of the coefficients of the current f, g in terms of the original f, g.

The rest of this section considers (1) constant-time computation, (2) "jumping" through division steps, and (3) further techniques to save time inside the computations.

5.2. Constant-time computation. The underlying Sage functions do not take constant time, but they can be replaced with functions that do take constant time. The case distinction can be replaced with constant-time bit operations, for example obtaining the new (δ, f, g) as follows:

```
def divstepsx(n,t,delta,f,g):
  assert t >= n and n >= 0
  f,g = f.truncate(t),g.truncate(t)
  kx = f.parent()
  x = kx.gen()
  u,v,q,r = kx(1),kx(0),kx(0),kx(1)

  while n > 0:
    f = f.truncate(t)
    if delta > 0 and g[0] != 0: delta,f,g,u,v,q,r = -delta,g,f,q,r,u,v
    f0,g0 = f[0],g[0]
    delta,g,q,r = 1+delta,(f0*g-g0*f)/x,(f0*q-g0*u)/x,(f0*r-g0*v)/x
    n,t = n-1,t-1
    g = kx(g).truncate(t)

  M2kx = MatrixSpace(kx.fraction_field(),2)
  return delta,f,g,M2kx((u,v,q,r))
```

Figure 5.1: Algorithm `divstepsx` to compute $(\delta_n, f_n, g_n) = \text{divstep}^n(\delta, f, g)$. Inputs: $n, t \in \mathbf{Z}$ with $0 \le n \le t$; $\delta \in \mathbf{Z}$; $f, g \in k[[x]]$ to precision at least t. Outputs: δ_n; f_n to precision t if $n = 0$, or $t - (n-1)$ if $n \ge 1$; g_n to precision $t - n$; $\mathcal{T}_{n-1} \cdots \mathcal{T}_0$.

- Let s be the "sign bit" of $-\delta$, meaning 0 if $-\delta \ge 0$ or 1 if $-\delta < 0$.
- Let z be the "nonzero bit" of $g(0)$, meaning 0 if $g(0) = 0$ or 1 if $g(0) \ne 0$.
- Compute and output $1 + (1 - 2sz)\delta$. For example, shift δ to obtain 2δ, "AND" with $-sz$ to obtain $2sz\delta$, and subtract from $1 + \delta$.
- Compute and output $f \oplus sz(f \oplus g)$, obtaining f if $sz = 0$ or g if $sz = 1$.
- Compute and output $(1 - 2sz)(f(0)g - g(0)f)/x$. Alternatively, compute and output simply $(f(0)g - g(0)f)/x$. See the discussion of decomposition and scaling in Section 3.

Standard techniques minimize the exact cost of these computations for various representations of the inputs. The details depend on the platform. See our case study in Section 7.

5.3. Jumping through division steps. Here is a more general divide-and-conquer algorithm to compute (δ_n, f_n, g_n) and the n-step transition matrix $\mathcal{T}_{n-1} \cdots \mathcal{T}_0$:

- If $n \le 1$, use the definition of divstep and stop.
- Choose $j \in \{1, 2, \ldots, n-1\}$.
- Jump j steps from δ, f, g to δ_j, f_j, g_j. Specifically, compute the j-step transition matrix $\mathcal{T}_{j-1} \cdots \mathcal{T}_0$, and then multiply by $\begin{pmatrix} f \\ g \end{pmatrix}$ to obtain $\begin{pmatrix} f_j \\ g_j \end{pmatrix}$. To compute the j-step transition matrix, call the same algorithm recursively. The critical point here is that this recursive call uses the inputs to precision only j: this determines the j-step transition matrix by Theorem 4.5.
- Similarly jump another $n - j$ steps from δ_j, f_j, g_j to δ_n, f_n, g_n.

```
from divstepsx import divstepsx

def jumpdivstepsx(n,t,delta,f,g):
  assert t >= n and n >= 0
  kx = f.truncate(t).parent()

  if n <= 1: return divstepsx(n,t,delta,f,g)

  j = n//2

  delta,f1,g1,P1 = jumpdivstepsx(j,j,delta,f,g)
  f,g = P1*vector((f,g))
  f,g = kx(f).truncate(t-j),kx(g).truncate(t-j)

  delta,f2,g2,P2 = jumpdivstepsx(n-j,n-j,delta,f,g)
  f,g = P2*vector((f,g))
  f,g = kx(f).truncate(t-n+1),kx(g).truncate(t-n)

  return delta,f,g,P2*P1
```

Figure 5.4: Algorithm `jumpdivstepsx` to compute $(\delta_n, f_n, g_n) = \text{divstep}^n(\delta, f, g)$. Same inputs and outputs as in Figure 5.1.

This splits an (n, t) problem into two smaller problems, namely a (j, j) problem and an $(n - j, n - j)$ problem, at the cost of $O(1)$ polynomial multiplications involving $O(t + n)$ coefficients.

If j is always chosen as 1 then this algorithm ends up performing the same computations as Figure 5.1: compute δ_1, f_1, g_1 to precision $t - 1$, compute δ_2, f_2, g_2 to precision $t - 2$, etc. But there are many more possibilities for j.

Our `jumpdivstepsx` algorithm in Figure 5.4 takes $j = \lfloor n/2 \rfloor$, balancing the (j, j) problem with the $(n - j, n - j)$ problem. In particular, with subquadratic polynomial multiplication, the algorithm is subquadratic. With FFT-based polynomial multiplication, the algorithm uses $(t + n)(\log(t + n))^{1+o(1)} + n(\log n)^{2+o(1)}$ operations, and hence $n(\log n)^{2+o(1)}$ operations if t is bounded by $n(\log n)^{1+o(1)}$. For comparison, Figure 5.1 takes $\Theta(n(t + n))$ operations.

We emphasize that, unlike standard fast-gcd algorithms such as [66], this algorithm does not call a polynomial-division subroutine between its two recursive calls.

5.5. More speedups. Our `jumpdivstepsx` algorithm is asymptotically $\Theta(\log n)$ times slower than fast polynomial multiplication. The best previous gcd algorithms also have a $\Theta(\log n)$ slowdown, both in the worst case and in the average case.[10] This might suggest that there is very little room for improvement.

However, Θ says nothing about constant factors. There are several standard ideas for constant-factor speedups in gcd computation, beyond lower-level speedups in multiplication. We now apply the ideas to `jumpdivstepsx`.

The final polynomial multiplications in `jumpdivstepsx` produce six large outputs: f, g, and four entries of a matrix product P. Callers do not always use all of these outputs: for example, the recursive calls to `jumpdivstepsx` do not use the resulting f

[10] There are faster cases. Strassen [66] gave an algorithm that replaces the log factor with the entropy of the list of degrees in the corresponding continued fraction; there are occasional inputs where this entropy is $o(\log n)$. For example, computing $\gcd\{\ldots, 0\}$ takes linear time. However, these speedups are not useful for constant-time computations.

and g. In principle there is no difficulty applying dead-value elimination and higher-level optimizations to each of the 63 possible combinations of desired outputs.

The recursive calls in `jumpdivstepsx` produce two products P_1, P_2 of transition matrices, which are then (if desired) multiplied to obtain $P = P_2P_1$. In Figure 5.4, `jumpdivstepsx` multiplies f and g first by P_1, and then by P_2, to obtain f_n and g_n. An alternative is to multiply by P.

The inputs f and g are truncated to t coefficients, and the resulting f_n and g_n are truncated to $t-n+1$ and $t-n$ coefficients respectively. Often the caller (for example, `jumpdivstepsx` itself) wants more coefficients of $P\begin{pmatrix} f \\ g \end{pmatrix}$. Rather than performing an entirely new multiplication by P, one can add the truncation errors back into the results: multiply P by the f and g truncation errors, multiply P_2 by the f_j and g_j truncation errors, and skip the final truncations of f_n and g_n.

There are standard speedups for multiplication in the context of truncated products, sums of products (e.g., in matrix multiplication), partially known products (e.g., the coefficients of $x^{-1}, x^{-2}, x^{-3}, \ldots$ in $P_1\begin{pmatrix} f \\ g \end{pmatrix}$ are known to be 0), repeated inputs (e.g., each entry of P_1 is multiplied by two entries of P_2 and by one of f, g), and outputs being reused as inputs. See, e.g., the discussions of "FFT addition", "FFT caching", and "FFT doubling" in [11].

Any choice of j between 1 and $n-1$ produces correct results. Values of j close to the edges do not take advantage of fast polynomial multiplication, but this does not imply that $j = \lfloor n/2 \rfloor$ is optimal. The number of combinations of choices of j and other options above is small enough that, for each small (t, n) in turn, one can simply measure all options and select the best. The results depend on the context—which outputs are needed—and on the exact performance of polynomial multiplication.

6 Fast polynomial gcd computation and modular inversion

This section presents three algorithms: `gcdx` computes $\gcd\{R_0, R_1\}$, where R_0 is a polynomial of degree $d > 0$ and R_1 is a polynomial of degree $< d$; `gcddegree` computes merely $\deg \gcd\{R_0, R_1\}$; `recipx` computes the reciprocal of R_1 modulo R_0 if $\gcd\{R_0, R_1\} = 1$. Figure 6.1 specifies all three algorithms, and Theorem 6.2 justifies all three algorithms. The theorem is proven in Appendix A.

Each algorithm calls `divstepsx` from Section 5 to compute $(\delta_{2d-1}, f_{2d-1}, g_{2d-1}) = \text{divstep}^{2d-1}(1, f, g)$. Here $f = x^d R_0(1/x)$ is the reversal of R_0, and $g = x^{d-1}R_1(1/x)$. Of course, one can replace `divstepsx` here with `jumpdivstepsx`, which has the same outputs. The algorithms vary in the input-precision parameter t passed to `divstepsx`, and vary in how they use the outputs:

- `gcddegree` uses input precision $t = 2d-1$ to compute δ_{2d-1}, and then uses one part of Theorem 6.2, which states that $\deg \gcd\{R_0, R_1\} = \delta_{2d-1}/2$.
- `gcdx` uses precision $t = 3d-1$ to compute δ_{2d-1} and $d+1$ coefficients of f_{2d-1}. The algorithm then uses another part of Theorem 6.2, which states that $f_{2d-1}/f_{2d-1}(0)$ is the reversal of $\gcd\{R_0, R_1\}$.
- `recipx` uses $t = 2d-1$ to compute δ_{2d-1}; 1 coefficient of f_{2d-1}; and the product P of transition matrices. It then uses the last part of Theorem 6.2, which (in the coprime case) states that the top-right corner of P is $f_{2d-1}(0)/x^{2d-2}$ times the degree-$(d-1)$ reversal of the desired reciprocal.

One can generalize to compute $\gcd\{R_0, R_1\}$ for any polynomials R_0, R_1 of degree $\le d$ in time that depends only on d: scan the inputs to find the top degree, and always perform

```
from divstepsx import divstepsx

def gcddegree(R0,R1):
  d = R0.degree()
  assert d > 0 and d > R1.degree()
  f,g = R0.reverse(d),R1.reverse(d-1)
  delta,f,g,P = divstepsx(2*d-1,2*d-1,1,f,g)
  return delta//2

def gcdx(R0,R1):
  d = R0.degree()
  assert d > 0 and d > R1.degree()
  f,g = R0.reverse(d),R1.reverse(d-1)
  delta,f,g,P = divstepsx(2*d-1,3*d-1,1,f,g)
  return f.reverse(delta//2)/f[0]

def recipx(R0,R1):
  d = R0.degree()
  assert d > 0 and d > R1.degree()
  f,g = R0.reverse(d),R1.reverse(d-1)
  delta,f,g,P = divstepsx(2*d-1,2*d-1,1,f,g)
  if delta != 0: return
  kx = f.parent()
  x = kx.gen()
  return kx(x^(2*d-2)*P[0][1]/f[0]).reverse(d-1)
```

Figure 6.1: Algorithm `gcdx` to compute $\gcd\{R_0, R_1\}$; algorithm `gcddegree` to compute $\deg \gcd\{R_0, R_1\}$; algorithm `recipx` to compute the reciprocal of R_1 modulo R_0 when $\gcd\{R_0, R_1\} = 1$. All three algorithms assume that $R_0, R_1 \in k[x]$, that $\deg R_0 > 0$, and that $\deg R_0 > \deg R_1$.

$2d$ iterations of divstep. The extra iteration handles the case that both R_0 and R_1 have degree d. For simplicity we focus on the case $\deg R_0 = d > \deg R_1$ with $d > 0$.

One can similarly characterize gcd and modular reciprocal in terms of subsequent iterates of divstep, with δ increased by the number of extra steps. Implementors might, for example, prefer to use an iteration count that is divisible by a large power of 2.

Theorem 6.2. *Let k be a field. Let d be a positive integer. Let R_0, R_1 be elements of the polynomial ring $k[x]$ with $\deg R_0 = d > \deg R_1$. Define $G = \gcd\{R_0, R_1\}$, and let V be the unique polynomial of degree $< d - \deg G$ such that $VR_1 \equiv G \pmod{R_0}$. Define $f = x^d R_0(1/x)$; $g = x^{d-1} R_1(1/x)$; $(\delta_n, f_n, g_n) = \text{divstep}^n(1, f, g)$; $\mathcal{T}_n = \mathcal{T}(\delta_n, f_n, g_n)$; and $\begin{pmatrix} u_n & v_n \\ q_n & r_n \end{pmatrix} = \mathcal{T}_{n-1} \cdots \mathcal{T}_0$. Then*

$$\begin{aligned} \deg G &= \delta_{2d-1}/2; \\ G &= x^{\deg G} f_{2d-1}(1/x)/f_{2d-1}(0); \\ V &= x^{-d+1+\deg G} v_{2d-1}(1/x)/f_{2d-1}(0). \end{aligned}$$

6.3. A numerical example. Take $k = \mathbf{F}_7$, $d = 7$, $R_0 = 2x^7 + 7x^6 + 1x^5 + 8x^4 + 2x^3 + 8x^2 + 1x + 8$, and $R_1 = 3x^6 + 1x^5 + 4x^4 + 1x^3 + 5x^2 + 9x + 2$. Then $f = 2 + 7x + 1x^2 + 8x^3 + 2x^4 + 8x^5 + 1x^6 + 8x^7$ and $g = 3 + 1x + 4x^2 + 1x^3 + 5x^4 + 9x^5 + 2x^6$. The `gcdx` algorithm computes $(\delta_{13}, f_{13}, g_{13}) = \text{divstep}^{13}(1, f, g) = (0, 2, 6)$; see Table 4.1

for all iterates of divstep on these inputs. Dividing f_{13} by its constant coefficient produces 1, the reversal of $\gcd\{R_0, R_1\}$. The `gcddegree` algorithm simply computes $\delta_{13} = 0$, which is enough information to see that $\gcd\{R_0, R_1\} = 1$.

6.4. Constant-factor speedups. Compared to the general power-series setting in `divstepsx`, the inputs to `gcddegree`, `gcdx`, and `recipx` are more limited. For example, the f input is all 0 after the first $d+1$ coefficients, so computations involving subsequent coefficients can be skipped. Similarly, the top-right entry of the matrix P in `recipx` is guaranteed to have coefficient 0 for 1, $1/x$, $1/x^2$, and so on through $1/x^{d-2}$, so one can save time in the polynomial computations that produce this entry. See Theorems A.1 and A.2 for bounds on the sizes of all intermediate results.

6.5. Bottom-up gcd and inversion. Our division steps eliminate bottom coefficients of the inputs. Appendices B, C, and D relate this to a traditional Euclid–Stevin computation, which gradually eliminates top coefficients of the inputs. The reversal of inputs and outputs in `gcddegree`, `gcdx`, and `recipx` exchanges the role of top coefficients and bottom coefficients. The following theorem states an alternative: skip the reversal, and instead directly eliminate the bottom coefficients of the original inputs.

Theorem 6.6. *Let k be a field. Let d be a positive integer. Let f, g be elements of the polynomial ring $k[x]$ with $f(0) \neq 0$, $\deg f \leq d$, and $\deg g < d$. Define $\gamma = \gcd\{f, g\}$. Define $(\delta_n, f_n, g_n) = \operatorname{divstep}^n(1, f, g)$; $\mathcal{T}_n = \mathcal{T}(\delta_n, f_n, g_n)$; and $\begin{pmatrix} u_n & v_n \\ q_n & r_n \end{pmatrix} = \mathcal{T}_{n-1} \cdots \mathcal{T}_0$. Then*

$$\deg \gamma = \deg f_{2d-1};$$
$$\gamma = f_{2d-1}/\ell;$$
$$x^{2d-2}\gamma \equiv x^{2d-2} v_{2d-1} g/\ell \pmod{f}$$

where ℓ is the leading coefficient of f_{2d-1}.

Proof. Note that $\gamma(0) \neq 0$, since f is a multiple of γ.

Define $R_0 = x^d f(1/x)$. Then $R_0 \in k[x]$; $\deg R_0 = d$ since $f(0) \neq 0$; and $f = x^d R_0(1/x)$.

Define $R_1 = x^{d-1} g(1/x)$. Then $R_1 \in k[x]$; $\deg R_1 < d$; and $g = x^{d-1} R_1(1/x)$.

Define $G = \gcd\{R_0, R_1\}$. We will show below that $\gamma = c x^{\deg G} G(1/x)$ for some $c \in k^*$. Then $\gamma = c f_{2d-1}/f_{2d-1}(0)$ by Theorem 6.2; i.e., γ is a constant multiple of f_{2d-1}. Hence $\deg \gamma = \deg f_{2d-1}$ as claimed. Furthermore γ has leading coefficient 1, so $1 = c\ell/f_{2d-1}(0)$; i.e., $\gamma = f_{2d-1}/\ell$ as claimed.

By Theorem 4.2, $f_{2d-1} = u_{2d-1} f + v_{2d-1} g$. Also $x^{2d-2} u_{2d-1}, x^{2d-2} v_{2d-1} \in k[x]$ by Theorem 4.3, so

$$x^{2d-2}\gamma = (x^{2d-2} u_{2d-1}/\ell) f + (x^{2d-2} v_{2d-1}/\ell) g \equiv (x^{2d-2} v_{2d-1}/\ell) g \pmod{f}$$

as claimed.

All that remains is to show that $\gamma = c x^{\deg G} G(1/x)$ for some $c \in k^*$. If $R_1 = 0$ then $G = \gcd\{R_0, 0\} = R_0/R_0[d]$ so $x^{\deg G} G(1/x) = x^d R_0(1/x)/R_0[d] = f/f[0]$; also $g = 0$ so $\gamma = f/f[\deg f] = (f[0]/f[\deg f]) x^{\deg G} G(1/x)$. Assume from now on that $R_1 \neq 0$.

We have $R_1 = QG$ for some $Q \in k[x]$. Note that $\deg R_1 = \deg Q + \deg G$. Also $R_1(1/x) = Q(1/x) G(1/x)$, so $x^{\deg R_1} R_1(1/x) = x^{\deg Q} Q(1/x) x^{\deg G} G(1/x)$ since $\deg R_1 = \deg Q + \deg G$. Hence $x^{\deg G} G(1/x)$ divides the polynomial $x^{\deg R_1} R_1(1/x)$, which in turn divides $x^{d-1} R_1(1/x) = g$. Similarly $x^{\deg G} G(1/x)$ divides f. Hence $x^{\deg G} G(1/x)$ divides $\gcd\{f, g\} = \gamma$.

Furthermore, $G = AR_0 + BR_1$ for some $A, B \in k[x]$. Take any integer m above all of $\deg G, \deg A, \deg B$; then

$$\begin{aligned} x^{m+d-\deg G} x^{\deg G} G(1/x) &= x^{m+d} G(1/x) \\ &= x^m A(1/x) x^d R_0(1/x) + x^{m+1} B(1/x) x^{d-1} R_1(1/x) \\ &= x^{m-\deg A} x^{\deg A} A(1/x) f + x^{m+1-\deg B} x^{\deg B} B(1/x) g, \end{aligned}$$

so $x^{m+d-\deg G}x^{\deg G}G(1/x)$ is a multiple of γ, so the polynomial $\gamma/x^{\deg G}G(1/x)$ divides $x^{m+d-\deg G}$, so $\gamma/x^{\deg G}G(1/x) = cx^i$ for some $c \in k^*$ and some $i \geq 0$. We must have $i = 0$ since $\gamma(0) \neq 0$. □

6.7. How to divide by x^{2d-2} modulo f. Unlike top-down inversion (and top-down gcd and bottom-up gcd), bottom-up inversion naturally scales its result by a power of x. We now explain various ways to remove this scaling.

For simplicity we focus here on the case $\gcd\{f, g\} = 1$. The divstep computation in Theorem 6.6 naturally produces the polynomial $x^{2d-2}v_{2d-1}/\ell$, which has degree at most $d-1$ by Theorem 6.2. Our objective is to divide this polynomial by x^{2d-2} modulo f, obtaining the reciprocal of g modulo f by Theorem 6.6. One way to do this is to multiply by a reciprocal of x^{2d-2} modulo f. This reciprocal depends only on d and f; i.e., it can be precomputed before g is available.

Here are several standard strategies to compute $1/x^{2d-2}$ modulo f, or more generally $1/x^e$ modulo f for any nonnegative integer e:

- Compute the polynomial $(1 - f/f(0))/x$, which is $1/x$ modulo f; and then raise this to the eth power modulo f. This takes a logarithmic number of multiplications modulo f.
- Start from $1/f(0)$, the reciprocal of f modulo x. Compute the reciprocals of f modulo x^2, x^4, x^8, etc. by Newton (Hensel) iteration. After finding the reciprocal r of f modulo x^e, divide $1 - rf$ by x^e to obtain the reciprocal of x^e modulo f. This takes a constant number of full-size multiplications, a constant number of half-size multiplications, etc. The total number of multiplications is again logarithmic, but if e is linear, as in the case $e = 2d - 2$, then the total size of the multiplications is linear *without* an extra logarithmic factor.
- Combine the powering strategy with the Hensel strategy. For example, use the reciprocal of f modulo x^{d-1} to compute the reciprocal of x^{d-1} modulo f, and then square modulo f to obtain the reciprocal of x^{2d-2} modulo f, rather than using the reciprocal of f modulo x^{2d-2}.
- Write down a simple formula for the reciprocal of x^e modulo f, if f has a special structure that makes this easy. For example, if $f = x^d - 1$ as in original NTRU and $d \geq 3$, then the reciprocal of x^{2d-2} is simply x^2. As another example, if $f = x^d + x^{d-1} + \cdots + 1$ and $d \geq 5$, then the reciprocal of x^{2d-2} is x^4.

In most cases the division by x^{2d-2} modulo f involves extra arithmetic that is avoided by the top-down reversal approach. On the other hand, the case $f = x^d - 1$ does not involve any extra arithmetic. There are also applications of inversion that can easily handle a scaled result.

7 Software case study for polynomial modular inversion

As a concrete case study, we consider the problem of inversion in the ring $(\mathbf{Z}/3)[x]/(x^{700} + x^{699} + \cdots + x + 1)$ on an Intel Haswell CPU core. The situation before our work was that this inversion consumed half of the key-generation time in the `ntruhrss701` cryptosystem, about 150000 cycles out of 300000 cycles. This cryptosystem was introduced by Hülsing, Rijneveld, Schanck, and Schwabe [39] at CHES 2017.[11]

[11] NTRU-HRSS is another round-2 submission in NIST's ongoing post-quantum competition. Google has also selected NTRU-HRSS for its "CECPQ2" post-quantum experiment. CECPQ2 includes a slightly modified version of `ntruhrss701`, and the NTRU-HRSS authors have also announced their intention to tweak NTRU-HRSS; these changes do not affect the performance of key generation.

We saved 60000 cycles out of these 150000 cycles, reducing the total key-generation time to 240000 cycles. Our approach also simplifies code, for example eliminating the series of conditional power-of-2 rotations in [39].

7.1. Details of the new computation. Our computation works with one integer δ and four polynomials $v, r, f, g \in (\mathbf{Z}/3)[x]$. Initially $\delta = 1$; $v = 0$; $r = 1$; $f = x^{700} + x^{699} + \cdots + x + 1$; and $g = g_{699}x^{699} + \cdots + g_0x^0$ is obtained by reversing the 700 input coefficients. We actually store $-\delta$ instead of δ; this turns out to be marginally more efficient for this platform.

We then apply $2 \cdot 700 - 1 = 1399$ iterations of a main loop. Each iteration works as follows, with the order of operations determined experimentally to try to minimize latency issues:

- Replace v with xv.
- Compute a swap mask as -1 if $\delta > 0$ and $g(0) \neq 0$, otherwise 0.
- Compute $c \in \mathbf{Z}/3$ as $f(0)g(0)$.
- Replace δ with $-\delta$ if the swap mask is set.
- Add 1 to δ.
- Replace (f, g) with (g, f) if the swap mask is set.
- Replace g with $(g - cf)/x$.
- Replace (v, r) with (r, v) if the swap mask is set.
- Replace r with $r - cv$.

This multiplies the transition matrix $\mathcal{T}(\delta, f, g)$ into the vector $\begin{pmatrix} v/x^{n-1} \\ r/x^n \end{pmatrix}$ where n is the number of iterations, and at the same time replaces (δ, f, g) with $\operatorname{divstep}(\delta, f, g)$. Actually, we are slightly modifying divstep here (and making the corresponding modification to $\mathcal{T}$), replacing the original coefficients $f(0)$ and $g(0)$ with the scaled coefficients 1 and $g(0)/f(0) = f(0)g(0)$, as in Section 3.4. In other words, if $f(0) = -1$, then we negate both f and g. We suppress further comments on this deviation from the definition of divstep.

The effect of n iterations is to replace the original (δ, f, g) with $\operatorname{divstep}^n(\delta, f, g)$. The matrix product $\mathcal{T}_{n-1} \cdots \mathcal{T}_0$ has the form $\begin{pmatrix} \cdots & v/x^{n-1} \\ \cdots & r/x^n \end{pmatrix}$. The new f and g are congruent to v/x^{n-1} and r/x^n respectively times the original g modulo $x^{700} + x^{699} + \cdots + x + 1$.

After 1399 iterations, we have $\delta = 0$ if and only if the input is invertible modulo $x^{700} + x^{699} + \cdots + x + 1$, by Theorem 6.2. Furthermore, if $\delta = 0$, then the inverse is $x^{-699}v_{1399}(1/x)/f(0)$ where $v_{1399} = v/x^{1398}$; i.e., the inverse is $x^{699}v(1/x)/f(0)$. We thus multiply v by $f(0)$, and take the coefficients of $x^0, \ldots, x^{699}$ in reverse order, to obtain the desired inverse.

We use signed representatives $-1, 0, 1$ for elements of $\mathbf{Z}/3$. We use 2-bit two's-complement representations of $-1, 0, 1$: the bottom bit is $1, 0, 1$ respectively, and the top bit is $1, 0, 0$. We wrote a simple superoptimizer to find optimal sequences of $3, 6, 6$ bit operations respectively for multiplication, addition, and subtraction. We do not claim novelty for the approach in this paragraph: Boothby and Bradshaw in [20, Section 4.1] suggested the same 2-bit representation for $\mathbf{Z}/3$ (without explaining it as signed two's-complement), and found the same operation counts by a similar search.

We store each of the four polynomials v, r, f, g as six 256-bit vectors: for example, the coefficients of $x^0, \ldots, x^{255}$ in v are stored as a 256-bit vector of bottom bits and a 256-bit vector of top bits. Within each 256-bit vector, we use the first 64-bit word to

store the coefficients of $x^0, x^4, \ldots, x^{252}$; the next 64-bit word to store the coefficients of $x^1, x^5, \ldots, x^{253}$; etc. Multiplying by x thus moves the first 64-bit word to the second, moves the second to the third, moves the third to the fourth, moves the fourth to the first, and shifts the first by 1 bit; the bit that falls off the end of the first word is inserted into the next vector.

The first 256 iterations involve only the first 256-bit vectors for v and r, so we simply leave the second and third vectors as 0. The next 256 iterations involve only the first two 256-bit vectors for v and r. Similarly, we manipulate only the first 256-bit vectors for f and g in the final 256 iterations, and we manipulate only the first and second 256-bit vectors for f and g in the previous 256 iterations. Our main loop thus has five sections: 256 iterations involving $(1, 1, 3, 3)$ vectors for (v, r, f, g); 256 iterations involving $(2, 2, 3, 3)$ vectors; 375 iterations involving $(3, 3, 3, 3)$ vectors; 256 iterations involving $(3, 3, 2, 2)$ vectors; and 256 iterations involving $(3, 3, 1, 1)$ vectors. This makes the code longer but saves almost 20% of the vector manipulations.

7.2. Comparison to the old computation. Many features of our computation are also visible in the software from [39]. We emphasize the ways that our computation is more streamlined.

There are essentially the same four polynomials v, r, f, g in [39] (labeled "c", "b", "g", "f"). Instead of a single integer δ (plus a loop counter), there are four integers "deg_f", "deg_g", "k", and "*done*" (plus a loop counter). One cannot simply eliminate "deg_f" and "deg_g" in favor of their difference δ, since "*done*" is set when "deg_f" reaches 0, and "*done*" influences "k", which in turn influences the results of the computation: the final result is multiplied by x^{701-k}.

Multiplication by x^{701-k} is a conceptually simple matter of rotating by k positions within 701 positions, and then reducing modulo $x^{700} + x^{699} + \cdots + x + 1$, since $x^{701} - 1$ is a multiple of $x^{700} + x^{699} + \cdots + x + 1$. However, since k is a variable, the obvious method of rotating by k positions does not take constant time; [39] instead performs conditional rotations by 1 position, 2 positions, 4 positions, etc., where the condition bits are the bits of k.

The inputs and outputs are not reversed in [39]. This affects the power of x multiplied into the final results. Our approach skips the powers of x entirely.

Each polynomial in [39] is represented as six 256-bit vectors, with the five-section pattern skipping manipulations of some vectors as explained above. The order of coefficients in each vector is simply $x^0, x^1, \ldots$; multiplication by x in [39] uses more arithmetic instructions than our approach does.

At a lower level, [39] uses unsigned representatives $0, 1, 2$ of $\mathbf{Z}/3$, and uses a manually designed sequence of 19 bit operations for a multiply-add operation in $\mathbf{Z}/3$. Langley [43] suggested a sequence of 16 bit operations, or 14 if an "and-not" operation is available. As in [20], we use just 9 bit operations.

The inversion software from [39] is 798 lines (32273 bytes) of Python generating assembly, producing 26634 bytes of object code. Our inversion software is 541 lines (14675 bytes) of C with intrinsics, compiled with `gcc` 8.2.1 to generate assembly, producing 10545 bytes of object code.

7.3. More NTRU examples. More than 1/3 of the key-generation time in [39] is consumed by a second inversion, which replaces the modulus 3 with 2^{13}. This is handled in [39] with an initial inversion modulo 2 followed by 8 multiplications modulo 2^{13} for a Newton iteration.[12] The initial inversion modulo 2 is handled in [39] by exponentiation,

[12] This use of Newton iteration follows the NTRU key-generation strategy suggested in [61], but we point out a difference in the details. All 8 multiplications in [39] are carried out modulo 2^{13}, while [61] instead follows the traditional precision doubling in Newton iteration: compute an inverse modulo 2^2, then 2^4, then 2^8 (or 2^7), then 2^{13}. Perhaps limiting the precision of the first 6 multiplications would save time in [39].

taking just 10332 cycles; the point here is that squaring modulo 2 is particularly efficient.

Much more inversion time is spent in key generation in `sntrup4591761`, a slightly larger cryptosystem from the NTRU Prime [14] family.[13] NTRU Prime uses a prime modulus, such as 4591 for `sntrup4591761`, to avoid concerns that a power-of-2 modulus could allow attacks (see generally [14]), but this modulus also makes arithmetic slower. Before our work, the key-generation software for `sntrup4591761` took 6 million cycles, partly for inversion in $(\mathbf{Z}/3)[x]/(x^{761}-x-1)$ but mostly for inversion in $(\mathbf{Z}/4591)[x]/(x^{761}-x-1)$. We reimplemented both of these inversion steps, reducing the total key-generation time to just 940852 cycles.[14] Almost all of our inversion code modulo 3 is shared between `sntrup4591761` and `ntruhrss701`.

7.4. Does lattice-based cryptography need fast inversion? Lyubashevsky, Peikert, and Regev [46] published an NTRU alternative that avoids inversions.[15] However, this cryptosystem has slower encryption than NTRU, has slower decryption than NTRU, and appears to be covered by a 2010 patent [35] by Gaborit and Aguilar-Melchor. It is easy to envision applications that will (1) select NTRU and (2) benefit from faster inversions in NTRU key generation.[16]

Lyubashevsky and Seiler have very recently announced "NTTRU" [47], a variant of NTRU with a ring chosen to allow very fast NTT-based (FFT-based) multiplication and division. This variant triggers many of the security concerns described in [14], but if the variant survives security analysis then it will eliminate concerns about key-generation speed in NTRU.

8 Definition of 2-adic division steps

Define $\operatorname{divstep} : \mathbf{Z} \times \mathbf{Z}_2^* \times \mathbf{Z}_2 \to \mathbf{Z} \times \mathbf{Z}_2^* \times \mathbf{Z}_2$ as follows:

$$\operatorname{divstep}(\delta, f, g) = \begin{cases} (1-\delta, g, (g-f)/2) & \text{if } \delta > 0 \text{ and } g \text{ is odd,} \\ (1+\delta, f, (g+(g \bmod 2)f)/2) & \text{otherwise.} \end{cases}$$

Note that $g-f$ is even in the first case (since both f and g are odd), and $g+(g \bmod 2)f$ is even in the second case (since f is odd).

8.1. Transition matrices. Write $(\delta_1, f_1, g_1) = \operatorname{divstep}(\delta, f, g)$. Then

$$\begin{pmatrix} f_1 \\ g_1 \end{pmatrix} = \mathcal{T}(\delta, f, g) \begin{pmatrix} f \\ g \end{pmatrix} \qquad \text{and} \qquad \begin{pmatrix} 1 \\ \delta_1 \end{pmatrix} = \mathcal{S}(\delta, f, g) \begin{pmatrix} 1 \\ \delta \end{pmatrix}$$

[13] NTRU Prime is another round-2 NIST submission. One other NTRU-based encryption system, NTRUEncrypt [29], was submitted to the competition, but has been merged into NTRU-HRSS for round 2; see [59]. For completeness we mention that NTRUEncrypt key generation took 980760 cycles for `ntrukem443` and 2639756 cycles for `ntrukem743`. As far as we know, the NTRUEncrypt software was not designed to run in constant time.

[14] This is a median cycle count from the SUPERCOP benchmarking framework. There is considerable variation in the cycle counts, more than 3% between quartiles, presumably depending on the mapping from virtual addresses to physical addresses. There is no dependence on the secret input being inverted.

[15] The LPR cryptosystem is the basis for several round-2 NIST submissions: Kyber, LAC, NewHope, Round5, Saber, ThreeBears, and the "NTRU LPRime" option in NTRU Prime.

[16] Google, for example, selected NTRU-HRSS for CECPQ2 as noted above, with the following explanation: "Schemes with a quotient-style key (like HRSS) will probably have faster encap/decap operations at the cost of much slower key-generation. Since there will be many uses outside TLS where keys can be reused, this is interesting as long as the key-generation speed is still reasonable for TLS." See [42].

where $\mathcal{T} : \mathbf{Z} \times \mathbf{Z}_2^* \times \mathbf{Z}_2 \to M_2(\mathbf{Z}[1/2])$ is defined by

$$\mathcal{T}(\delta, f, g) = \begin{cases} \begin{pmatrix} 0 & 1 \\ \frac{-1}{2} & \frac{1}{2} \end{pmatrix} & \text{if } \delta > 0 \text{ and } g \text{ is odd,} \\ \begin{pmatrix} 1 & 0 \\ \frac{g \bmod 2}{2} & \frac{1}{2} \end{pmatrix} & \text{otherwise,} \end{cases}$$

and $\mathcal{S} : \mathbf{Z} \times \mathbf{Z}_2^* \times \mathbf{Z}_2 \to M_2(\mathbf{Z})$ is defined by

$$\mathcal{S}(\delta, f, g) = \begin{cases} \begin{pmatrix} 1 & 0 \\ 1 & -1 \end{pmatrix} & \text{if } \delta > 0 \text{ and } g \text{ is odd,} \\ \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} & \text{otherwise.} \end{cases}$$

Note that both $\mathcal{T}(\delta, f, g)$ and $\mathcal{S}(\delta, f, g)$ are defined entirely by δ, the bottom bit of f (which is always 1), and the bottom bit of g; they do not depend on the remaining bits of f and g.

8.2. Decomposition. This 2-adic divstep, like the power-series divstep, is a composition of two simpler functions. Specifically, it is a conditional swap that replaces (δ, f, g) with $(-\delta, g, -f)$ if $\delta > 0$ and g is odd, followed by an elimination that replaces (δ, f, g) with $(1 + \delta, f, (g + (g \bmod 2)f)/2)$.

8.3. Sizes. Write $(\delta_1, f_1, g_1) = \text{divstep}(\delta, f, g)$. If f and g are integers that fit into $n + 1$ bits two's-complement, in the sense that $-2^n \le f < 2^n$ and $-2^n \le g < 2^n$, then also $-2^n \le f_1 < 2^n$ and $-2^n \le g_1 < 2^n$. Similarly, if $-2^n < f < 2^n$ and $-2^n < g < 2^n$ then $-2^n < f_1 < 2^n$ and $-2^n < g_1 < 2^n$. Beware, however, that intermediate results such as $g - f$ need an extra bit.

As in the polynomial case, an important feature of divstep is that iterating divstep on integer inputs eventually sends g to 0. Concretely, we will show that $(D + o(1))n$ iterations are sufficient, where $D = 2/(10 - \log_2 633) = 2.882100569\ldots$; see Theorem 11.2 for exact bounds. The proof technique cannot do better than $(d + o(1))n$ iterations, where $d = 14/(15 - \log_2(561 + \sqrt{249185})) = 2.828339631\ldots$. Numerical evidence is consistent with the possibility that $(d + o(1))n$ iterations are required in the worst case, which is what matters in the context of constant-time algorithms.

8.4. A non-functioning variant. Many variations are possible in the definition of divstep. However, some care is required, as the following variant illustrates.

Define posdivstep : $\mathbf{Z} \times \mathbf{Z}_2^* \times \mathbf{Z}_2 \to \mathbf{Z} \times \mathbf{Z}_2^* \times \mathbf{Z}_2$ as follows:

$$\text{posdivstep}(\delta, f, g) = \begin{cases} (1 - \delta, g, (g + f)/2) & \text{if } \delta > 0 \text{ and } g \text{ is odd,} \\ (1 + \delta, f, (g + (g \bmod 2)f)/2) & \text{otherwise.} \end{cases}$$

This is just like divstep except that it eliminates the negation of f in the swapped case.

It is not true that iterating posdivstep on integer inputs eventually sends g to 0. For example, $\text{posdivstep}^2(1, 1, 1) = (1, 1, 1)$. For comparison, in the polynomial case we were free to negate f (or g) at any moment.

8.5. A centered variant. As another variant of divstep, we define cdivstep : $\mathbf{Z} \times \mathbf{Z}_2^* \times \mathbf{Z}_2 \to \mathbf{Z} \times \mathbf{Z}_2^* \times \mathbf{Z}_2$ as follows:

$$\text{cdivstep}(\delta, f, g) = \begin{cases} (1 - \delta, g, (f - g)/2) & \text{if } \delta > 0 \text{ and } g \text{ is odd,} \\ (1 + \delta, f, (g + (g \bmod 2)f)/2) & \text{if } \delta = 0, \\ (1 + \delta, f, (g - (g \bmod 2)f)/2) & \text{otherwise.} \end{cases}$$

Iterating cdivstep produces the same intermediate results as a variable-time gcd algorithm introduced by Stehlé and Zimmermann in [62]. The analogous gcd algorithm for divstep is a new variant of the Stehlé–Zimmermann algorithm, and we analyze the performance of this variant to prove bounds on the performance of divstep; see Appendices E, F, and G. As an analogy, one of our proofs in the polynomial case relies on relating x-adic divstep to the Euclid–Stevin algorithm.

The extra case distinctions make each cdivstep iteration more complicated than each divstep iteration. Similarly, the Stehlé–Zimmermann algorithm is more complicated than the new variant. To understand the motivation for cdivstep, compare $\text{divstep}^3(-2,f,g)$ to $\text{cdivstep}^3(-2,f,g)$. One has $\text{divstep}^3(-2,f,g) = (1,f,g_3)$ where

$$8g_3 \in \{g, g+f, g+2f, g+3f, g+4f, g+5f, g+6f, g+7f\}.$$

One has $\text{cdivstep}^3(-2,f,g) = (1,f,g'_3)$ where

$$8g'_3 \in \{g-3f, g-2f, g-f, g, g+f, g+2f, g+3f, g+4f\}.$$

It seems intuitively clear that the "centered" output g'_3 is smaller than the "uncentered" output g_3; that, more generally, cdivstep has smaller outputs than divstep; and that cdivstep thus uses fewer iterations than divstep.

However, our analyses do not support this intuition. All available evidence indicates that, surprisingly, the worst case of cdivstep uses almost 10% more iterations than the worst case of divstep. Concretely, our proof technique cannot do better than $(c+o(1))n$ iterations for cdivstep, where $c = 2/(\log_2(\sqrt{17}-1)-1) = 3.110510062\ldots$, and numerical evidence is consistent with the possibility that $(c+o(1))n$ iterations are required.

8.6. A plus-or-minus variant. As yet another example, the following variant of divstep is essentially[17] the Brent–Kung "Algorithm PM" from [24]:

$$\text{pmdivstep}(\delta,f,g) = \begin{cases} (-\delta, g, (f-g)/2) & \text{if } \delta \geq 0 \text{ and } (g-f) \bmod 4 = 0, \\ (-\delta, g, (f+g)/2) & \text{if } \delta \geq 0 \text{ and } (g+f) \bmod 4 = 0, \\ (\delta, f, (g-f)/2) & \text{if } \delta < 0 \text{ and } (g-f) \bmod 4 = 0, \\ (\delta, f, (g+f)/2) & \text{if } \delta < 0 \text{ and } (g+f) \bmod 4 = 0, \\ (1+\delta, f, g/2) & \text{otherwise.} \end{cases}$$

There are even more cases here than in cdivstep, although splitting out an initial conditional swap reduces the number of cases to 3. Another complication here is that the linear combinations used in pmdivstep depend on the bottom *two* bits of f and g rather than just the bottom bit.

To understand the motivation for pmdivstep, note that after each addition or subtraction there are always at least two halvings, as in the "sliding window" approach to exponentiation. Again it seems intuitively clear that this reduces the number of iterations required. Consider, however, the following example (which is from [24, Theorem 3] modulo minor details): pmdivstep needs $3b-1$ iterations to reach $g=0$ starting from $(1,1,3\cdot 2^{b-2})$. Specifically, the first $b-2$ iterations produce

$$(2,1,3\cdot 2^{b-3}), (3,1,3\cdot 2^{b-4}), \ldots, (b-2,1,3\cdot 2), (b-1,1,3),$$

and then the next $2b+1$ iterations produce

$$(1-b,3,2), (2-b,3,1), (2-b,3,2), (3-b,3,1), (3-b,3,2), \ldots,$$
$$(-1,3,1), (-1,3,2), (0,3,1), (0,1,2), (1,1,1), (-1,1,0).$$

[17] The Brent–Kung algorithm has an extra loop to allow the case that both inputs f,g are even. Compare [24, Figure 4] to the definition of pmdivstep.

Brent and Kung prove that $\lceil cn \rceil$ systolic cells are enough for their algorithm, where $c = 3.110510062\ldots$ is the constant mentioned above, and they conjecture that $(3 + o(1))n$ cells are enough, so presumably $(3 + o(1))n$ iterations of pmdivstep are enough. Our analysis shows that fewer iterations are sufficient for divstep, even though divstep is simpler than pmdivstep.

9 Iterates of 2-adic division steps

The following results are analogous to the x-adic results in Section 4. We state these results separately to support verification.

Starting from $(\delta, f, g) \in \mathbf{Z} \times \mathbf{Z}_2^* \times \mathbf{Z}_2$, define $(\delta_n, f_n, g_n) = \text{divstep}^n(\delta, f, g) \in \mathbf{Z} \times \mathbf{Z}_2^* \times \mathbf{Z}_2$; $\mathcal{T}_n = \mathcal{T}(\delta_n, f_n, g_n) \in M_2(\mathbf{Z}[1/2])$; and $\mathcal{S}_n = \mathcal{S}(\delta_n, f_n, g_n) \in M_2(\mathbf{Z})$.

Theorem 9.1. $\begin{pmatrix} f_n \\ g_n \end{pmatrix} = \mathcal{T}_{n-1} \cdots \mathcal{T}_m \begin{pmatrix} f_m \\ g_m \end{pmatrix}$ *and* $\begin{pmatrix} 1 \\ \delta_n \end{pmatrix} = \mathcal{S}_{n-1} \cdots \mathcal{S}_m \begin{pmatrix} 1 \\ \delta_m \end{pmatrix}$ *if* $n \geq m \geq 0$.

Proof. This follows from $\begin{pmatrix} f_{m+1} \\ g_{m+1} \end{pmatrix} = \mathcal{T}_m \begin{pmatrix} f_m \\ g_m \end{pmatrix}$ and $\begin{pmatrix} 1 \\ \delta_{m+1} \end{pmatrix} = \mathcal{S}_m \begin{pmatrix} 1 \\ \delta_m \end{pmatrix}$. □

Theorem 9.2. $\mathcal{T}_{n-1} \cdots \mathcal{T}_m \in \begin{pmatrix} \frac{1}{2^{n-m-1}}\mathbf{Z} & \frac{1}{2^{n-m-1}}\mathbf{Z} \\ \frac{1}{2^{n-m}}\mathbf{Z} & \frac{1}{2^{n-m}}\mathbf{Z} \end{pmatrix}$ *if* $n > m \geq 0$.

Proof. As in Theorem 4.3. □

Theorem 9.3. $\mathcal{S}_{n-1} \cdots \mathcal{S}_m \in \begin{pmatrix} 1 & 0 \\ \{2-(n-m), \ldots, n-m-2, n-m\} & \{1, -1\} \end{pmatrix}$ *if* $n > m \geq 0$.

Proof. As in Theorem 4.4. □

Theorem 9.4. *Let* m, t *be nonnegative integers. Assume that* $\delta'_m = \delta_m$*;* $f'_m \equiv f_m \pmod{2^t}$*; and* $g'_m \equiv g_m \pmod{2^t}$*. Then, for each integer* n *with* $m < n \leq m + t$*:* $\mathcal{T}'_{n-1} = \mathcal{T}_{n-1}$*;* $\mathcal{S}'_{n-1} = \mathcal{S}_{n-1}$*;* $\delta'_n = \delta_n$*;* $f'_n \equiv f_n \pmod{2^{t-(n-m-1)}}$*; and* $g'_n \equiv g_n \pmod{2^{t-(n-m)}}$.

Proof. Fix m, t and induct on n. Observe that $(\delta'_{n-1}, f'_{n-1} \bmod 2, g'_{n-1} \bmod 2)$ equals $(\delta_{n-1}, f_{n-1} \bmod 2, g_{n-1} \bmod 2)$:

- If $n = m + 1$: By assumption $f'_m \equiv f_m \pmod{2^t}$, and $n \leq m + t$ so $t \geq 1$, so in particular $f'_m \bmod 2 = f_m \bmod 2$. Similarly $g'_m \bmod 2 = g_m \bmod 2$. By assumption $\delta'_m = \delta_m$.
- If $n > m + 1$: $f'_{n-1} \equiv f_{n-1} \pmod{2^{t-(n-m-2)}}$ by the inductive hypothesis, and $n \leq m+t$ so $t-(n-m-2) \geq 2$, so in particular $f'_{n-1} \bmod 2 = f_{n-1} \bmod 2$; similarly $g'_{n-1} \equiv g_{n-1} \pmod{2^{t-(n-m-1)}}$ by the inductive hypothesis, and $t-(n-m-1) \geq 1$, so in particular $g'_{n-1} \bmod 2 = g_{n-1} \bmod 2$; and $\delta'_{n-1} = \delta_{n-1}$ by the inductive hypothesis.

Now use the fact that $\mathcal{T}$ and $\mathcal{S}$ inspect only the bottom bit of each number to see that

$$\mathcal{T}'_{n-1} = \mathcal{T}(\delta'_{n-1}, f'_{n-1}, g'_{n-1}) = \mathcal{T}(\delta_{n-1}, f_{n-1}, g_{n-1}) = \mathcal{T}_{n-1},$$
$$\mathcal{S}'_{n-1} = \mathcal{S}(\delta'_{n-1}, f'_{n-1}, g'_{n-1}) = \mathcal{S}(\delta_{n-1}, f_{n-1}, g_{n-1}) = \mathcal{S}_{n-1}$$

as claimed.

Multiply $\begin{pmatrix} 1 \\ \delta'_{n-1} \end{pmatrix} = \begin{pmatrix} 1 \\ \delta_{n-1} \end{pmatrix}$ by $\mathcal{S}'_{n-1} = \mathcal{S}_{n-1}$ to see that $\delta'_n = \delta_n$ as claimed.

```
def truncate(f,t):
  if t == 0: return 0
  twot = 1<<(t-1)
  return ((f+twot)&(2*twot-1))-twot

def divsteps2(n,t,delta,f,g):
  assert t >= n and n >= 0
  f,g = truncate(f,t),truncate(g,t)
  u,v,q,r = 1,0,0,1

  while n > 0:
    f = truncate(f,t)
    if delta > 0 and g&1: delta,f,g,u,v,q,r = -delta,g,-f,q,r,-u,-v
    g0 = g&1
    delta,g,q,r = 1+delta,(g+g0*f)/2,(q+g0*u)/2,(r+g0*v)/2
    n,t = n-1,t-1
    g = truncate(ZZ(g),t)

  M2Q = MatrixSpace(QQ,2)
  return delta,f,g,M2Q((u,v,q,r))
```

Figure 10.1: Algorithm `divsteps2` to compute $(\delta_n, f_n, g_n) = \text{divstep}^n(\delta, f, g)$. Inputs: $n, t \in \mathbf{Z}$ with $0 \le n \le t$; $\delta \in \mathbf{Z}$; at least bottom t bits of $f, g \in \mathbf{Z}_2$. Outputs: δ_n; bottom t bits of f_n if $n = 0$, or $t - (n-1)$ bits if $n \ge 1$; bottom $t - n$ bits of g_n; $\mathcal{T}_{n-1} \cdots \mathcal{T}_0$.

By the inductive hypothesis, $(\mathcal{T}'_m, \ldots, \mathcal{T}'_{n-2}) = (\mathcal{T}_m, \ldots, \mathcal{T}_{n-2})$, and $\mathcal{T}'_{n-1} = \mathcal{T}_{n-1}$, so $\mathcal{T}'_{n-1} \cdots \mathcal{T}'_m = \mathcal{T}_{n-1} \cdots \mathcal{T}_m$. Abbreviate $\mathcal{T}_{n-1} \cdots \mathcal{T}_m$ as P. Then $\begin{pmatrix} f'_n \\ g'_n \end{pmatrix} = P \begin{pmatrix} f'_m \\ g'_m \end{pmatrix}$ and $\begin{pmatrix} f_n \\ g_n \end{pmatrix} = P \begin{pmatrix} f_m \\ g_m \end{pmatrix}$ by Theorem 9.1, so $\begin{pmatrix} f'_n - f_n \\ g'_n - g_n \end{pmatrix} = P \begin{pmatrix} f'_m - f_m \\ g'_m - g_m \end{pmatrix}$.

By Theorem 9.2, $P \in \begin{pmatrix} \frac{1}{2^{n-m-1}}\mathbf{Z} & \frac{1}{2^{n-m-1}}\mathbf{Z} \\ \frac{1}{2^{n-m}}\mathbf{Z} & \frac{1}{2^{n-m}}\mathbf{Z} \end{pmatrix}$. By assumption, $f'_m - f_m$ and $g'_m - g_m$ are multiples of 2^t. Multiply to see that $f'_n - f_n$ is a multiple of $2^t/2^{n-m-1} = 2^{t-(n-m-1)}$ as claimed, and $g'_n - g_n$ is a multiple of $2^t/2^{n-m} = 2^{t-(n-m)}$ as claimed. □

10 Fast computation of iterates of 2-adic division steps

We describe just as in Section 5 two algorithms to compute $(\delta_n, f_n, g_n) = \text{divstep}^n(\delta, f, g)$. We are given the bottom t bits of f, g and compute f_n, g_n to $t-n+1, t-n$ bits respectively if $n \ge 1$; see Theorem 9.4. We also compute the product of the relevant $\mathcal{T}$ matrices. We specify these algorithms in Figures 10.1–10.2 as functions `divsteps2` and `jumpdivsteps2` in Sage.

The first function `divsteps2` simply takes divsteps one after another, analogously to `divstepsx`. The second function `jumpdivsteps2` uses a straightforward divide-and-conquer strategy, analogously to `jumpdivstepsx`. More generally, j in `jumpdivsteps2` can be taken anywhere between 1 and $n-1$; this generalization includes `divsteps2` as a special case.

As in Section 5, the Sage functions are not constant-time, but it is easy to build constant-time versions of the same computations. The comments on performance in Section 5 are applicable here, with integer arithmetic replacing polynomial arithmetic. The same basic optimization techniques are also applicable here.

```
from divsteps2 import divsteps2,truncate

def jumpdivsteps2(n,t,delta,f,g):
  assert t >= n and n >= 0
  if n <= 1: return divsteps2(n,t,delta,f,g)

  j = n//2

  delta,f1,g1,P1 = jumpdivsteps2(j,j,delta,f,g)
  f,g = P1*vector((f,g))
  f,g = truncate(ZZ(f),t-j),truncate(ZZ(g),t-j)

  delta,f2,g2,P2 = jumpdivsteps2(n-j,n-j,delta,f,g)
  f,g = P2*vector((f,g))
  f,g = truncate(ZZ(f),t-n+1),truncate(ZZ(g),t-n)

  return delta,f,g,P2*P1
```

Figure 10.2: Algorithm `jumpdivsteps2` to compute $(\delta_n, f_n, g_n) = \text{divstep}^n(\delta, f, g)$. Same inputs and outputs as in Figure 10.1.

11 Fast integer gcd computation and modular inversion

This section presents two algorithms. If f is an odd integer and g is an integer then `gcd2` computes $\gcd\{f, g\}$. If also $\gcd\{f, g\} = 1$ then `recip2` computes the reciprocal of g modulo f. Figure 11.1 specifies both algorithms, and Theorem 11.2 justifies both algorithms.

The algorithms take the smallest nonnegative integer d such that $|f| < 2^d$ and $|g| < 2^d$. More generally, one can take d as an extra input; the algorithms then take constant time if d is constant. Theorem 11.2 relies on $f^2 + 4g^2 \leq 5 \cdot 2^{2d}$ (which is certainly true if $|f| < 2^d$ and $|g| < 2^d$), but does not rely on d being minimal. One can further generalize to allow even f, by first finding the number of shared powers of 2 in f and g and then reducing to the odd case.

The algorithms then choose a positive integer m as a particular function of d, and call `divsteps2` (which can be transparently replaced with `jumpdivsteps2`) to compute $(\delta_m, f_m, g_m) = \text{divstep}^m(1, f, g)$. The choice of m guarantees $g_m = 0$ and $f_m = \pm\gcd\{f, g\}$ by Theorem 11.2.

The `gcd2` algorithm uses input precision $m+d$ to obtain $d+1$ bits of f_m, i.e., to obtain the signed remainder of f_m modulo 2^{d+1}. This signed remainder is exactly $f_m = \pm\gcd\{f, g\}$, since the assumptions $|f| < 2^d$ and $|g| < 2^d$ imply $|\gcd\{f, g\}| < 2^d$.

The `recip2` algorithm uses input precision just $m+1$ to obtain the signed remainder of f_m modulo 4. This algorithm assumes $\gcd\{f, g\} = 1$, so $f_m = \pm 1$, so the signed remainder is exactly f_m. The algorithm also extracts the top-right corner v_m of the transition matrix, and multiplies by f_m, obtaining a reciprocal of g modulo f by Theorem 11.2.

Both `gcd2` and `recip2` are bottom-up algorithms, and in particular `recip2` naturally obtains this reciprocal $v_m f_m$ as a fraction with denominator 2^{m-1}. It multiplies by 2^{m-1} to obtain an integer, and then multiplies by $((f+1)/2)^{m-1}$ modulo f to divide by 2^{m-1} modulo f. The reciprocal of 2^{m-1} can be computed ahead of time. Another way to compute this reciprocal is through Hensel lifting to compute $f^{-1} \pmod{2^{m-1}}$; for details and further techniques see Section 6.7.

We emphasize that `recip2` assumes that its inputs are coprime; it does not notice if this assumption is violated. One can check the assumption by computing f_m to higher precision (as in `gcd2`), or by multiplying the supposed reciprocal by g and seeing whether

```
from divsteps2 import divsteps2

def iterations(d):
  return (49*d+80)//17 if d<46 else (49*d+57)//17

def gcd2(f,g):
  assert f & 1
  d = max(f.nbits(),g.nbits())
  m = iterations(d)
  delta,fm,gm,P = divsteps2(m,m+d,1,f,g)
  return abs(fm)

def recip2(f,g):
  assert f & 1
  d = max(f.nbits(),g.nbits())
  m = iterations(d)
  precomp = Integers(f)((f+1)/2)^(m-1)
  delta,fm,gm,P = divsteps2(m,m+1,1,f,g)
  V = sign(fm)*ZZ(P[0][1]*2^(m-1))
  return ZZ(V*precomp)
```

Figure 11.1: Algorithm `gcd2` to compute $\gcd\{f, g\}$; algorithm `recip2` to compute the reciprocal of g modulo f when $\gcd\{f, g\} = 1$. Both algorithms assume that f is odd.

the result is 1 modulo f. For comparison, the `recip` algorithm from Section 6 *does* notice if its polynomial inputs are coprime, using a quick test of δ_m.

Theorem 11.2. *Let f be an odd integer. Let g be an integer. Define $(\delta_n, f_n, g_n) = \text{divstep}^n(1, f, g)$; $\mathcal{T}_n = \mathcal{T}(\delta_n, f_n, g_n)$; and $\begin{pmatrix} u_n & v_n \\ q_n & r_n \end{pmatrix} = \mathcal{T}_{n-1} \cdots \mathcal{T}_0$. Let d be a real number. Assume that $f^2 + 4g^2 \leq 5 \cdot 2^{2d}$. Let m be an integer. Assume that $m \geq \lfloor (49d + 80)/17 \rfloor$ if $d < 46$, and that $m \geq \lfloor (49d + 57)/17 \rfloor$ if $d \geq 46$. Then $m \geq 1$; $g_m = 0$; $f_m = \pm \gcd\{f, g\}$; $2^{m-1} v_m \in \mathbf{Z}$; and $2^{m-1} v_m g \equiv 2^{m-1} f_m \pmod{f}$.*

In particular, if $\gcd\{f, g\} = 1$, then $f_m = \pm 1$, and dividing $2^{m-1} v_m f_m$ by 2^{m-1} modulo f produces the reciprocal of g modulo f.

Proof. First $f^2 \geq 1$ so $1 \leq f^2 + 4g^2 \leq 5 \cdot 2^{2d} < 2^{2d+7/3}$ since $5^3 < 2^7$. Hence $d > -7/6$. If $d < 46$ then $m \geq \lfloor (49d + 80)/17 \rfloor \geq \lfloor (49(-7/6) + 80)/17 \rfloor = 1$. If $d \geq 46$ then $m \geq \lfloor (49d + 57)/17 \rfloor \geq \lfloor (49 \cdot 46 + 57)/17 \rfloor = 135$. Either way $m \geq 1$ as claimed.

Define $R_0 = f$, $R_1 = 2g$, and $G = \gcd\{R_0, R_1\}$. Then $G = \gcd\{f, g\}$ since f is odd.

Define $b = \log_2 \sqrt{R_0^2 + R_1^2} = \log_2 \sqrt{f^2 + 4g^2} \geq 0$. Then $2^{2b} = f^2 + 4g^2 \leq 5 \cdot 2^{2d}$ so $b \leq d + \log_4 5$. We now split into three cases, in each case defining n as in Theorem G.6:

- If $b \leq 21$: Define $n = \lfloor 19b/7 \rfloor$. Then $n \leq \lfloor 49b/17 \rfloor \leq \lfloor 49(d + \log_4 5)/17 \rfloor \leq \lfloor (49d + 57)/17 \rfloor \leq m$ since $5^{49} \leq 4^{57}$.
- If $21 < b \leq 46$: Define $n = \lfloor (49b + 23)/17 \rfloor$. If $d \geq 46$ then $n \leq \lfloor (49d + 23)/17 \rfloor \leq \lfloor (49d + 57)/17 \rfloor \leq m$. Otherwise $d < 46$ so $n \leq \lfloor (49(d + \log_4 5) + 23)/17 \rfloor \leq \lfloor (49d + 80)/17 \rfloor \leq m$.
- If $b > 46$: Define $n = \lfloor 49b/17 \rfloor$. Then $n \leq \lfloor 49b/17 \rfloor \leq \lfloor 49(d + \log_4 5)/17 \rfloor \leq \lfloor (49d + 57)/17 \rfloor \leq m$.

In all cases $n \leq m$.

Now $\text{divstep}^n(1, R_0, R_1/2) \in \mathbf{Z} \times \mathbf{Z} \times \{0\}$ by Theorem G.6, so $\text{divstep}^m(1, R_0, R_1/2) \in \mathbf{Z} \times \mathbf{Z} \times \{0\}$, so

$$(\delta_m, f_m, g_m) = \text{divstep}^m(1, f, g) = \text{divstep}^m(1, R_0, R_1/2) \in \mathbf{Z} \times \{G, -G\} \times \{0\}$$

by Theorem G.3. Hence $g_m = 0$ as claimed, and $f_m = \pm G = \pm \gcd\{f, g\}$ as claimed.

Both u_m and v_m are in $(1/2^{m-1})\mathbf{Z}$ by Theorem 9.2. In particular, $2^{m-1}v_m \in \mathbf{Z}$ as claimed. Finally, $f_m = u_m f + v_m g$ by Theorem 9.1, so $2^{m-1}f_m = 2^{m-1}u_m f + 2^{m-1}v_m g$, and $2^{m-1}u_m \in \mathbf{Z}$, so $2^{m-1}f_m \equiv 2^{m-1}v_m g \pmod{f}$ as claimed. □

12 Software case study for integer modular inversion

As emphasized in Section 1, we have selected an inversion problem to be extremely favorable to Fermat's method. We use Intel platforms with 64-bit multipliers. We focus on the problem of inverting modulo the well-known Curve25519 prime $p = 2^{255} - 19$. There has been a long line of Curve25519 implementation papers starting with [10] in 2006; we compare to the latest speed records [54] (in assembly) for inversion modulo this prime.

Despite all these Fermat advantages, we achieve better speeds using our 2-adic division steps. As mentioned in Section 1, we take 10050 cycles, 8778 cycles, and 8543 cycles on Haswell, Skylake, and Kaby Lake, where [54] used 11854 cycles, 9301 cycles, and 8971 cycles. This section looks more closely at how the new computation works.

12.1. Details of the new computation. Theorem 11.2 guarantees that 738 divstep iterations suffice, since $\lfloor (49 \cdot 255 + 57)/17 \rfloor = 738$. To simplify the computation, we actually use $744 = 12 \cdot 62$ iterations.

The decisions in the first 62 iterations are determined entirely by the bottom 62 bits of the inputs. We perform these iterations entirely in 64-bit words; apply the resulting 62-step transition matrix to the entire original inputs, to jump to the result of 62 divstep iterations; and then repeat.

This is similar in two important ways to Lehmer's gcd computation from [44]. One of Lehmer's ideas, mentioned before, is to carry out some steps in limited precision. Another of Lehmer's ideas is for this limit to be a small constant. Our advantage over Lehmer's computation is that we have a completely regular constant-time data flow.

There are many other possibilities for which precision to use at each step. All of these possibilities can be described using the jumping framework described in Section 5.3, which applies to both the x-adic case and the 2-adic case. Figure 12.2 gives three examples of jump strategies. The first strategy computes each divstep in turn to full precision, as in the case study in Section 7. The second strategy is what we use in this case study. The third strategy illustrates an asymptotically faster choice of jump distances. The third strategy might be more efficient than the second strategy in hardware, but the second strategy seems optimal for 64-bit CPUs.

In more detail, we invert a 255-bit x modulo p as follows:

1. Set $f = p$, $g = x$, $\delta = 1$, $i = 1$.
2. Set $f_0 = f \pmod{2^{64}}$, $g_0 = g \pmod{2^{64}}$.
3. Compute $(\delta', f_1, g_1) = \text{divstep}^{62}(\delta, f_0, g_0)$ and obtain a scaled transition matrix $\mathcal{T}_i$ s.t. $\frac{\mathcal{T}_i}{2^{62}} \begin{pmatrix} f_0 \\ g_0 \end{pmatrix} = \begin{pmatrix} f_1 \\ g_1 \end{pmatrix}$. The 63-bit signed entries of $\mathcal{T}_i$ fit into 64-bit registers. We call this step `jump64divsteps2`.
4. Compute $(f, g) \leftarrow \mathcal{T}_i(f, g)/2^{62}$. Set $\delta = \delta'$.

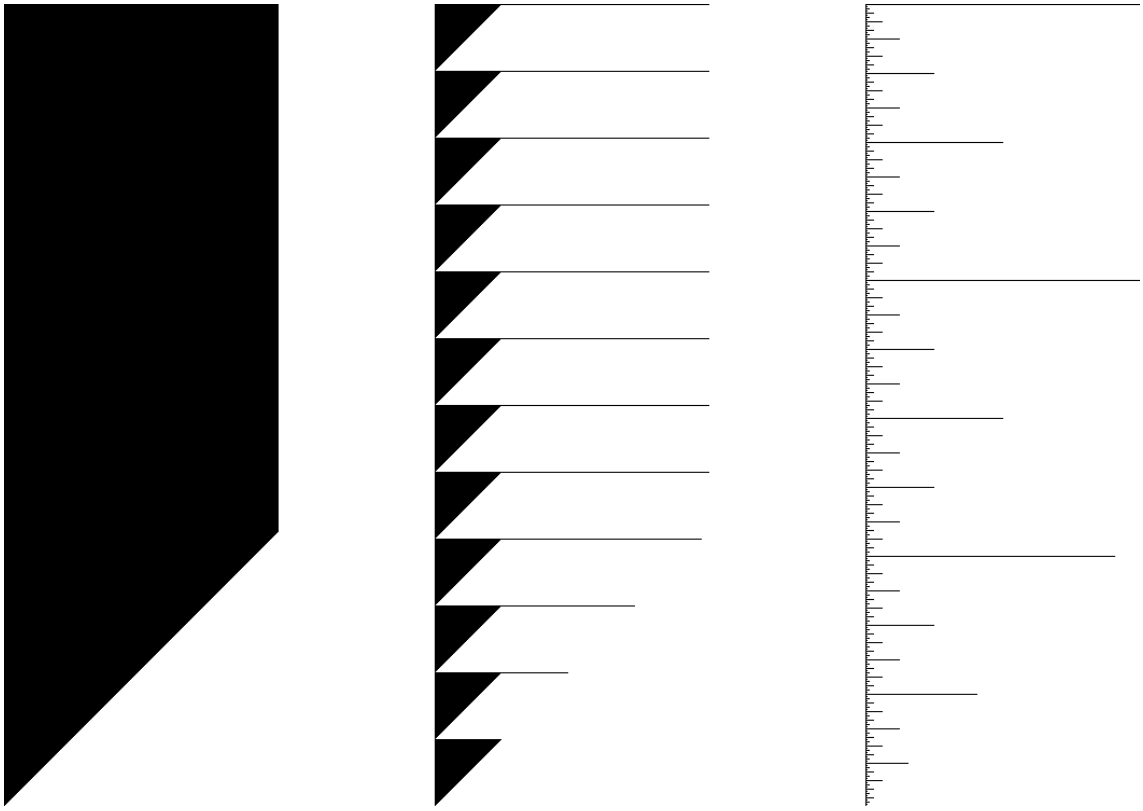

Figure 12.2: Three jump strategies for 744 divstep iterations on 255-bit integers. Left strategy: $j = 1$; i.e., computing each divstep in turn to full precision. Middle strategy: $j = 1$ for ≤ 62 requested iterations, else $j = 62$; i.e., using 62 bottom bits to compute 62 iterations, jumping 62 iterations, using 62 new bottom bits to compute 62 more iterations, etc. Right strategy: $j = 1$ for 2 requested iterations, $j = 2$ for 3 or 4 requested iterations, $j = 4$ for 5 or 6 or 7 or 8 requested iterations, etc. Vertical axis, 0 on top through 744 on bottom: number of iterations. Horizontal axis (within each strategy), 0 on left through 254 on right: bit positions used in computation.

5. Set $i \leftarrow i + 1$. Go back to step 3 if $i \leq 12$.
6. Compute $v \bmod p$ where v is the top-right corner of $\mathcal{T}_{12}\mathcal{T}_{11}\cdots\mathcal{T}_1$:
   (a) Compute pair-products $\mathcal{T}_{2i}\mathcal{T}_{2i-1}$ with entries being 126-bit signed integers which fits into two 64-bit limbs (radix 2^{64}).
   (b) Compute quad-products $\mathcal{T}_{4i}\mathcal{T}_{4i-1}\mathcal{T}_{4i-2}\mathcal{T}_{4i-3}$ with entries being 252-bit signed integers (four 64-bit limbs, radix 2^{64}).
   (c) At this point the three quad-products are converted into unsigned integers modulo p divided into 4 64-bit limbs.
   (d) Compute final vector-matrix-vector multiplications modulo p.
7. Compute $x^{-1} = \operatorname{sgn}(f) \cdot v \cdot 2^{-744} \pmod p$ where 2^{-744} is precomputed.

12.3. Other CPUs. Our advantage is larger on platforms with smaller multipliers. For example, we have written software to invert modulo $2^{255} - 19$ on an ARM Cortex-A7, obtaining a median cycle count of 35277 cycles. The best previous Cortex-A7 result we have found in the literature is 62648 cycles reported by Fujii and Aranha [34], using Fermat's method. Internally, our software does repeated calls to `jump32divsteps2`, units

of 30 iterations with the resulting transition matrix fitting inside 32-bit general-purpose registers.

12.4. Other primes. Nath and Sarkar present inversion speeds for 20 different special primes, 12 of which were considered in previous work. For concreteness we consider inversion modulo the double-size prime $2^{511} - 187$. Aranha–Barreto–Pereira–Ricardini [8] selected this prime for their "M-511" curve.

Nath and Sarkar report that their Fermat inversion software for $2^{511} - 187$ takes 72804 Haswell cycles, 47062 Skylake cycles, or 45014 Kaby Lake cycles. The slowdown from $2^{255} - 19$, more than $5\times$ in each case, is easy to explain: the exponent is twice as long, producing almost twice as many multiplications; each multiplication handles twice as many bits; and multiplication cost per bit is not constant.

Our 511-bit software is solidly faster, under 30000 cycles on all of these platforms. Most of the cycles in our computation are in evaluating `jump64divsteps2`, and the number of calls to `jump64divsteps2` scales linearly with the number of bits in the input. There is some overhead for multiplications, so a modulus of twice the size takes more than twice as long, but our scalability to large moduli is much better than the Fermat scalability.

Our advantage is also larger in applications that use "random" primes rather than special primes: we pay the extra cost for Montgomery multiplication only at the end of our computation, whereas Fermat inversion pays the extra cost in each step.

References

[1] — (no editor), *Actes du congrès international des mathématiciens, tome 3*, Gauthier-Villars Éditeur, Paris, 1971. See [41].

[2] — (no editor), *CWIT 2011: 12th Canadian workshop on information theory: Kelowna, British Columbia, May 17–20, 2011*, Institute of Electrical and Electronics Engineers, 2011. ISBN 978-1-4577-0743-8. See [51].

[3] Carlisle Adams, Jan Camenisch (editors), *Selected areas in cryptography—SAC 2017, 24th international conference, Ottawa, ON, Canada, August 16–18, 2017, revised selected papers*, Lecture Notes in Computer Science, 10719, Springer, 2018. ISBN 978-3-319-72564-2. See [15].

[4] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, *Hamming Quasi-Cyclic (HQC)* (2017). URL: https://pqc-hqc.org/doc/hqc-specification_2017-11-30.pdf. Citations in this document: §1.

[5] Alfred V. Aho (chairman), *Proceedings of fifth annual ACM symposium on theory of computing: Austin, Texas, April 30–May 2, 1973*, ACM, New York, 1973. See [53].

[6] Martin Albrecht, Carlos Cid, Kenneth G. Paterson, CJ Tjhai, Martin Tomlinson, *NTS-KEM*, "The main document submitted to NIST" (2017). URL: https://nts-kem.io/. Citations in this document: §1.

[7] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, Billy Bob Brumley, *Cache-timing attacks on RSA key generation* (2018). URL: https://eprint.iacr.org/2018/367. Citations in this document: §1.

[8] Diego F. Aranha, Paulo S. L. M. Barreto, Geovandro C. C. F. Pereira, Jefferson E. Ricardini, *A note on high-security general-purpose elliptic curves* (2013). URL: https://eprint.iacr.org/2013/647. Citations in this document: §12.4.

[9] Elwyn R. Berlekamp, *Algebraic coding theory*, McGraw-Hill, 1968. Citations in this document: §1.

[10] Daniel J. Bernstein, *Curve25519: new Diffie-Hellman speed records*, in PKC 2006 [70] (2006), 207–228. URL: https://cr.yp.to/papers.html#curve25519. Citations in this document: §1, §12.

[11] Daniel J. Bernstein, *Fast multiplication and its applications*, in [27] (2008), 325–384. URL: https://cr.yp.to/papers.html#multapps. Citations in this document: §5.5.

[12] Daniel J. Bernstein, Tung Chou, Tanja Lange, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Wen Wang, *Classic McEliece: conservative code-based cryptography*, "Supporting Documentation" (2017). URL: https://classic.mceliece.org/nist.html. Citations in this document: §1.

[13] Daniel J. Bernstein, Tung Chou, Peter Schwabe, *McBits: fast constant-time code-based cryptography*, in CHES 2013 [18] (2013), 250–272. URL: https://binary.cr.yp.to/mcbits.html. Citations in this document: §1, §1.

[14] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, Christine van Vredendaal, *NTRU Prime: reducing attack surface at low cost*, full version of [15] (2017). URL: https://ntruprime.cr.yp.to/papers.html. Citations in this document: §1, §1, §1.1, §7.3, §7.3, §7.4.

[15] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, Christine van Vredendaal, *NTRU Prime: reducing attack surface at low cost*, in SAC 2017 [3], abbreviated version of [14] (2018), 235–260.

[16] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, Bo-Yin Yang, *High-speed high-security signatures*, Journal of Cryptographic Engineering **2** (2012), 77–89; see also older version at CHES 2011. URL: https://ed25519.cr.yp.to/ed25519-20110926.pdf. Citations in this document: §1.

[17] Daniel J. Bernstein, Peter Schwabe, *NEON crypto*, in CHES 2012 [56] (2012), 320–339. URL: https://cr.yp.to/papers.html#neoncrypto. Citations in this document: §1, §1.

[18] Guido Bertoni, Jean-Sébastien Coron (editors), *Cryptographic hardware and embedded systems—CHES 2013—15th international workshop, Santa Barbara, CA, USA, August 20–23, 2013, proceedings*, Lecture Notes in Computer Science, 8086, Springer, 2013. ISBN 978-3-642-40348-4. See [13].

[19] Adam W. Bojanczyk, Richard P. Brent, *A systolic algorithm for extended GCD computation*, Computers & Mathematics with Applications **14** (1987), 233–238. ISSN 0898-1221. URL: https://maths-people.anu.edu.au/~brent/pd/rpb096i.pdf. Citations in this document: §1.3, §1.3.

[20] Tomas J. Boothby and Robert W. Bradshaw, *Bitslicing and the method of four Russians over larger finite fields* (2009). URL: http://arxiv.org/abs/0901.1413. Citations in this document: §7.1, §7.2.

[21] Joppe W. Bos, *Constant time modular inversion*, Journal of Cryptographic Engineering **4** (2014), 275–281. URL: http://joppebos.com/files/CTInversion.pdf. Citations in this document: §1, §1, §1, §1, §1, §1.3.

[22] Richard P. Brent, Fred G. Gustavson, David Y. Y. Yun, *Fast solution of Toeplitz systems of equations and computation of Padé approximants*, Journal of Algorithms **1** (1980), 259–295. ISSN 0196-6774. URL: https://maths-people.anu.edu.au/~brent/pd/rpb059i.pdf. Citations in this document: §1.4, §1.4.

[23] Richard P. Brent, Hsiang-Tsung Kung, *Systolic VLSI arrays for polynomial GCD computation*, IEEE Transactions on Computers **C-33** (1984), 731–736. URL: https://maths-people.anu.edu.au/~brent/pd/rpb073i.pdf. Citations in this document: §1.3, §1.3, §3.2.

[24] Richard P. Brent, Hsiang-Tsung Kung, *A systolic VLSI array for integer GCD computation*, ARITH-7 (1985), 118–125. URL: https://maths-people.anu.edu.au/~brent/pd/rpb077i.pdf. Citations in this document: §1.3, §1.3, §1.3, §1.3, §1.4, §8.6, §8.6, §8.6.

[25] Richard P. Brent, Paul Zimmermann, *An $O(M(n)\log n)$ algorithm for the Jacobi symbol*, in ANTS 2010 [37] (2010), 83–95. URL: https://arxiv.org/pdf/1004.2091.pdf. Citations in this document: §1.4.

[26] Duncan A. Buell (editor), *Algorithmic number theory, 6th international symposium, ANTS-VI, Burlington, VT, USA, June 13–18, 2004, proceedings*, Lecture Notes in Computer Science, 3076, Springer, 2004. ISBN 3-540-22156-5. See [62].

[27] Joe P. Buhler, Peter Stevenhagen (editors), *Surveys in algorithmic number theory*, Mathematical Sciences Research Institute Publications, 44, Cambridge University Press, New York, 2008. See [11].

[28] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, Joost Renes, *CSIDH: an efficient post-quantum commutative group action*, in Asiacrypt 2018 [55] (2018), 395–427. URL: https://csidh.isogeny.org/csidh-20181118.pdf. Citations in this document: §1.

[29] Cong Chen, Jeffrey Hoffstein, William Whyte, Zhenfei Zhang, *NIST PQ Submission: NTRUEncrypt: A lattice based encryption algorithm* (2017). URL: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions. Citations in this document: §7.3.

[30] Tung Chou, *McBits revisited*, in CHES 2017 [33] (2017), 213–231. URL: https://tungchou.github.io/mcbits/. Citations in this document: §1.

[31] Benoît Daireaux, Véronique Maume-Deschamps, Brigitte Vallée, *The Lyapunov tortoise and the dyadic hare*, in [49] (2005), 71–94. URL: http://emis.ams.org/journals/DMTCS/pdfpapers/dmAD0108.pdf. Citations in this document: §E.5, §F.2.

[32] Jean Louis Dornstetter, *On the equivalence between Berlekamp's and Euclid's algorithms*, IEEE Transactions on Information Theory **33** (1987), 428–431. Citations in this document: §1.

[33] Wieland Fischer, Naofumi Homma (editors), *Cryptographic hardware and embedded systems—CHES 2017—19th international conference, Taipei, Taiwan, September 25–28, 2017, proceedings*, Lecture Notes in Computer Science, 10529, Springer, 2017. ISBN 978-3-319-66786-7. See [30], [39].

[34] Hayato Fujii, Diego F. Aranha, *Curve25519 for the Cortex-M4 and beyond*, LATINCRYPT 2017, to appear (2017). URL: https://www.lasca.ic.unicamp.br/media/publications/paper39.pdf. Citations in this document: §1.1, §12.3.

[35] Philippe Gaborit, Carlos Aguilar Melchor, *Cryptographic method for communicating confidential information*, patent EP2537284B1 (2010). URL: https://patents.google.com/patent/EP2537284B1/en. Citations in this document: §7.4.

[36] Mariya Georgieva, Frédéric de Portzamparc, *Toward secure implementation of McEliece decryption*, in COSADE 2015 [48] (2015), 141–156. URL: https://eprint.iacr.org/2015/271. Citations in this document: §1.

[37] Guillaume Hanrot, François Morain, Emmanuel Thomé (editors), *Algorithmic number theory, 9th international symposium, ANTS-IX, Nancy, France, July 19–23, 2010, proceedings*, Lecture Notes in Computer Science, 6197, 2010. ISBN 978-3-642-14517-9. See [25].

[38] Agnes E. Heydtmann, Jørn M. Jensen, *On the equivalence of the Berlekamp-Massey and the Euclidean algorithms for decoding*, IEEE Transactions on Information Theory **46** (2000), 2614–2624. Citations in this document: §1.

[39] Andreas Hülsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, *High-speed key encapsulation from NTRU*, in CHES 2017 [33] (2017), 232–252. URL: https://eprint.iacr.org/2017/667. Citations in this document: §1, §1.1, §1.1, §1.3, §7, §7, §7.2, §7.2, §7.2, §7.2, §7.2, §7.2, §7.2, §7.2, §7.3, §7.3, §7.3, §7.3, §7.3.

[40] Burton S. Kaliski, *The Montgomery inverse and its applications*, IEEE Transactions on Computers **44** (1995), 1064–1065. Citations in this document: §1.

[41] Donald E. Knuth, *The analysis of algorithms*, in [1] (1971), 269–274. Citations in this document: §1, §1.4, §1.4.

[42] Adam Langley, *CECPQ2* (2018). URL: https://www.imperialviolet.org/2018/12/12/cecpq2.html. Citations in this document: §7.4.

[43] Adam Langley, *Add initial HRSS support* (2018). URL: https://boringssl.googlesource.com/boringssl/+/7b935937b18215294e7dbf6404742855e3349092/crypto/hrss/hrss.c. Citations in this document: §7.2.

[44] Derrick H. Lehmer, *Euclid's algorithm for large numbers*, American Mathematical Monthly **45** (1938), 227–233. ISSN 0002-9890. URL: http://links.jstor.org/sici?sici=0002-9890(193804)45:4<227:EAFLN>2.0.CO;2-Y. Citations in this document: §1, §1.4, §12.1.

[45] Xianhui Lu, Yamin Liu, Dingding Jia, Haiyang Xue, Jingnan He, Zhenfei Zhang, *LAC: lattice-based cryptosystems* (2017). URL: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions. Citations in this document: §1.

[46] Vadim Lyubashevsky, Chris Peikert, Oded Regev, *On ideal lattices and learning with errors over rings*, Journal of the ACM **60** (2013), Article 43, 35 pages. URL: https://eprint.iacr.org/2012/230. Citations in this document: §7.4.

[47] Vadim Lyubashevsky, Gregor Seiler, *NTTRU: truly fast NTRU using NTT* (2019). URL: https://eprint.iacr.org/2019/040. Citations in this document: §7.4.

[48] Stefan Mangard, Axel Y. Poschmann (editors), *Constructive side-channel analysis and secure design—6th international workshop, COSADE 2015, Berlin, Germany, April 13–14, 2015, revised selected papers*, Lecture Notes in Computer Science, 9064, Springer, 2015. ISBN 978-3-319-21475-7. See [36].

[49] Conrado Martínez (editor), *2005 international conference on analysis of algorithms: papers from the conference (AofA'05) held in Barcelona, June 6–10, 2005*, The Association. Discrete Mathematics & Theoretical Computer Science (DMTCS), Nancy, 2005. URL: http://www.emis.ams.org/journals/DMTCS/proceedings/dmAD01ind.html. See [31].

[50] James Massey, *Shift-register synthesis and BCH decoding*, IEEE Transactions on Information Theory **15** (1969), 122–127. ISSN 0018-9448. Citations in this document: §1.

[51] Todd D. Mateer, *On the equivalence of the Berlekamp-Massey and the euclidean algorithms for algebraic decoding*, in CWIT 2011 [2] (2011), 139–142. Citations in this document: §1.

[52] Niels Möller, *On Schönhage's algorithm and subquadratic integer GCD computation*, Mathematics of Computation **77** (2008), 589–607. URL: https://www.lysator.liu.se/~nisse/archive/S0025-5718-07-02017-0.pdf. Citations in this document: §1.4.

[53] Robert T. Moenck, *Fast computation of GCDs*, in STOC 1973 [5] (1973), 142–151. Citations in this document: §1.4, §1.4, §1.4, §1.4.

[54] Kaushik Nath, Palash Sarkar, *Efficient inversion in (pseudo-)Mersenne prime order fields* (2018). URL: https://eprint.iacr.org/2018/985. Citations in this document: §1.1, §12, §12.

[55] Thomas Peyrin, Steven D. Galbraith, *Advances in cryptology—ASIACRYPT 2018—24th international conference on the theory and application of cryptology and information security, Brisbane, QLD, Australia, December 2–6, 2018, proceedings, part I*, Lecture Notes in Computer Science, 11272, Springer, 2018. ISBN 978-3-030-03325-5. See [28].

[56] Emmanuel Prouff, Patrick Schaumont (editors), *Cryptographic hardware and embedded systems—CHES 2012—14th international workshop, Leuven, Belgium, September 9–12, 2012, proceedings*, Lecture Notes in Computer Science, 7428, Springer, 2012. ISBN 978-3-642-33026-1. See [17].

[57] Martin Roetteler, Michael Naehrig, Krysta M. Svore, Kristin E. Lauter, *Quantum resource estimates for computing elliptic curve discrete logarithms*, in ASIACRYPT 2017 [67] (2017), 241–270. URL: https://eprint.iacr.org/2017/598. Citations in this document: §1.1, §1.3.

[58] The Sage Developers (editor), *SageMath, the Sage Mathematics Software System (Version 8.0)*, 2017. URL: https://www.sagemath.org. Citations in this document: §1.2, §1.2, §2.2, §5.

[59] John Schanck, *Announcement of NTRU-HRSS-KEM and NTRUEncrypt merger* (2018). URL: https://groups.google.com/a/list.nist.gov/d/msg/pqc-forum/SrFO_vK3xbI/mSmjY0HZCgAJ. Citations in this document: §7.3.

[60] Arnold Schönhage, *Schnelle Berechnung von Kettenbruchentwicklungen*, Acta Informatica **1** (1971), 139–144. ISSN 0001-5903. Citations in this document: §1, §1.4, §1.4.

[61] Joseph H. Silverman, *Almost inverses and fast NTRU key creation*, NTRU Cryptosystems (Technical Note#014) (1999). URL: https://assets.securityinnovation.com/static/downloads/NTRU/resources/NTRUTech014.pdf. Citations in this document: §7.3, §7.3.

[62] Damien Stehlé, Paul Zimmermann, *A binary recursive gcd algorithm*, in ANTS 2004 [26] (2004), 411–425. URL: https://perso.ens-lyon.fr/damien.stehle/BINARY.html. Citations in this document: §1.4, §1.4, §8.5, §E, §E.6, §E.6, §E.6, §F.2, §F.2, §F.2, §H, §H.

[63] Josef Stein, *Computational problems associated with Racah algebra*, Journal of Computational Physics **1** (1967), 397–405. Citations in this document: §1.

[64] Simon Stevin, *L'arithmétique*, Imprimerie de Christophle Plantin, 1585. URL: http://www.dwc.knaw.nl/pub/bronnen/Simon_Stevin-[II_B]_The_Principal_Works_of_Simon_Stevin,_Mathematics.pdf. Citations in this document: §1.

[65] Volker Strassen, *The computational complexity of continued fractions*, in SYM-SAC 1981 [69] (1981), 51–67; see also newer version [66].

[66] Volker Strassen, *The computational complexity of continued fractions*, SIAM Journal on Computing **12** (1983), 1–27; see also older version [65]. ISSN 0097-5397. Citations in this document: §1.4, §1.4, §5.3, §5.5.

[67] Tsuyoshi Takagi, Thomas Peyrin (editors), *Advances in cryptology—ASIACRYPT 2017—23rd international conference on the theory and applications of cryptology and information security, Hong Kong, China, December 3–7, 2017, proceedings, part II*, Lecture Notes in Computer Science, 10625, Springer, 2017. ISBN 978-3-319-70696-2. See [57].

[68] Emmanuel Thomé, *Subquadratic computation of vector generating polynomials and improvement of the block Wiedemann algorithm*, Journal of Symbolic Computation **33** (2002), 757–775. URL: https://hal.inria.fr/inria-00103417/document. Citations in this document: §1.4.

[69] Paul S. Wang (editor), *SYM-SAC '81: proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Computation, Snowbird, Utah, August 5–7, 1981*, Association for Computing Machinery, New York, 1981. ISBN 0-89791-047-8. See [65].

[70] Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, Tal Malkin (editors), *Public key cryptography—9th international conference on theory and practice in public-key cryptography, New York, NY, USA, April 24–26, 2006, proceedings*, Lecture Notes in Computer Science, 3958, Springer, 2006. ISBN 978-3-540-33851-2. See [10].

A Proof of main gcd theorem for polynomials

We give two separate proofs of the facts stated earlier relating gcd and modular reciprocal to a particular iterate of divstep in the polynomial case:

- The second proof consists of a review of the Euclid–Stevin gcd algorithm (Appendix B); a description of exactly how the intermediate results in the Euclid–Stevin algorithm relate to iterates of divstep (Appendix C); and, finally, a translation of gcd/reciprocal results from the Euclid–Stevin context to the divstep context (Appendix D).

- The first proof, given in this appendix, is shorter: it works directly with divstep, without a detour through the Euclid–Stevin labeling of intermediate results.

Theorem A.1. *Let k be a field. Let d be a positive integer. Let R_0, R_1 be elements of the polynomial ring $k[x]$ with $\deg R_0 = d > \deg R_1$. Define $f = x^d R_0(1/x)$; $g = x^{d-1} R_1(1/x)$; and $(\delta_n, f_n, g_n) = \operatorname{divstep}^n(1, f, g)$ for $n \geq 0$. Then*

$$2\deg f_n \leq 2d - 1 - n + \delta_n \quad \text{for } n \geq 0,$$
$$2\deg g_n \leq 2d - 1 - n - \delta_n \quad \text{for } n \geq 0.$$

Proof. Induct on n. If $n = 0$ then $(\delta_n, f_n, g_n) = (1, f, g)$ so $2\deg f_n = 2\deg f \leq 2d = 2d - 1 - n + \delta_n$ and $2\deg g_n = 2\deg g \leq 2(d-1) = 2d - 1 - n - \delta_n$.

Assume from now on that $n \geq 1$. By the inductive hypothesis, $2\deg f_{n-1} \leq 2d - n + \delta_{n-1}$, and $2\deg g_{n-1} \leq 2d - n - \delta_{n-1}$.

Case 1: $\delta_{n-1} \leq 0$. Then

$$(\delta_n, f_n, g_n) = (1 + \delta_{n-1}, f_{n-1}, (f_{n-1}(0)g_{n-1} - g_{n-1}(0)f_{n-1})/x).$$

Both f_{n-1} and g_{n-1} have degree at most $(2d - n - \delta_{n-1})/2$, so the polynomial $xg_n = f_{n-1}(0)g_{n-1} - g_{n-1}(0)f_{n-1}$ also has degree at most $(2d - n - \delta_{n-1})/2$, so $2\deg g_n \leq 2d - n - \delta_{n-1} - 2 = 2d - 1 - n - \delta_n$. Also $2\deg f_n = 2\deg f_{n-1} \leq 2d - 1 - n + \delta_n$.

Case 2: $\delta_{n-1} > 0$ and $g_{n-1}(0) = 0$. Then

$$(\delta_n, f_n, g_n) = (1 + \delta_{n-1}, f_{n-1}, f_{n-1}(0)g_{n-1}/x),$$

so $2\deg g_n = 2\deg g_{n-1} - 2 \leq 2d - n - \delta_{n-1} - 2 = 2d - 1 - n - \delta_n$. As before $2\deg f_n = 2\deg f_{n-1} \leq 2d - 1 - n + \delta_n$.

Case 3: $\delta_{n-1} > 0$ and $g_{n-1}(0) \neq 0$. Then

$$(\delta_n, f_n, g_n) = (1 - \delta_{n-1}, g_{n-1}, (g_{n-1}(0)f_{n-1} - f_{n-1}(0)g_{n-1})/x).$$

This time the polynomial $xg_n = g_{n-1}(0)f_{n-1} - f_{n-1}(0)g_{n-1}$ also has degree at most $(2d - n + \delta_{n-1})/2$, so $2\deg g_n \leq 2d - n + \delta_{n-1} - 2 = 2d - 1 - n - \delta_n$. Also $2\deg f_n = 2\deg g_{n-1} \leq 2d - n - \delta_{n-1} = 2d - 1 - n + \delta_n$. □

Theorem A.2. *In the situation of Theorem A.1, define $\mathcal{T}_n = \mathcal{T}(\delta_n, f_n, g_n)$ and*

$$\begin{pmatrix} u_n & v_n \\ q_n & r_n \end{pmatrix} = \mathcal{T}_{n-1} \cdots \mathcal{T}_0.$$

Then $x^n(u_n r_n - v_n q_n) \in k^*$ for $n \geq 0$; $x^{n-1}u_n \in k[x]$ for $n \geq 1$; $x^{n-1}v_n, x^n q_n, x^n r_n \in k[x]$ for $n \geq 0$; and

$$2\deg(x^{n-1}u_n) \leq n - 3 + \delta_n \quad \text{for } n \geq 1,$$
$$2\deg(x^{n-1}v_n) \leq n - 1 + \delta_n \quad \text{for } n \geq 0,$$
$$2\deg(x^n q_n) \leq n - 1 - \delta_n \quad \text{for } n \geq 0,$$
$$2\deg(x^n r_n) \leq n + 1 - \delta_n \quad \text{for } n \geq 0.$$

Proof. Each matrix $\mathcal{T}_n$ has determinant in $(1/x)k^*$ by construction, so the product of n such matrices has determinant in $(1/x^n)k^*$. In particular, $u_n r_n - v_n q_n \in (1/x^n)k^*$.

Induct on n. For $n = 0$ we have $\delta_n = 1$, $u_n = 1$, $v_n = 0$, $q_n = 0$, $r_n = 1$ so $x^{n-1}v_n = 0 \in k[x]$; $x^n q_n = 0 \in k[x]$; $x^n r_n = 1 \in k[x]$; $2\deg(x^{n-1}v_n) = -\infty \leq n - 1 + \delta_n$; $2\deg(x^n q_n) = -\infty \leq n - 1 - \delta_n$; and $2\deg(x^n r_n) = 0 \leq n + 1 - \delta_n$.

Assume from now on that $n \geq 1$. By Theorem 4.3, $u_n, v_n \in (1/x^{n-1})k[x]$ and $q_n, r_n \in (1/x^n)k[x]$, so all that remains is to prove the degree bounds.

Case 1: $\delta_{n-1} \leq 0$. Then $\delta_n = 1 + \delta_{n-1}$; $u_n = u_{n-1}$; $v_n = v_{n-1}$; $q_n = (f_{n-1}(0)q_{n-1} - g_{n-1}(0)u_{n-1})/x$; and $r_n = (f_{n-1}(0)r_{n-1} - g_{n-1}(0)v_{n-1})/x$.

Note that $n > 1$ since $\delta_0 > 0$. By the inductive hypothesis, $2\deg(x^{n-2}u_{n-1}) \le n-4+\delta_{n-1}$, so $2\deg(x^{n-1}u_n) \le n-2+\delta_{n-1} = n-3+\delta_n$. Also $2\deg(x^{n-2}v_{n-1}) \le n-2+\delta_{n-1}$, so $2\deg(x^{n-1}v_n) \le n+\delta_{n-1} = n-1+\delta_n$.

For q_n, note that both $x^{n-1}q_{n-1}$ and $x^{n-1}u_{n-1}$ have degree at most $(n-2-\delta_{n-1})/2$, so the same is true for their k-linear combination $x^n q_n$. Hence $2\deg(x^n q_n) \le n-2-\delta_{n-1} = n-1-\delta_n$. Similarly $2\deg(x^n r_n) \le n-\delta_{n-1} = n+1-\delta_n$.

Case 2: $\delta_{n-1} > 0$ and $g_{n-1}(0) = 0$. Then $\delta_n = 1+\delta_{n-1}$; $u_n = u_{n-1}$; $v_n = v_{n-1}$; $q_n = f_{n-1}(0)q_{n-1}/x$; and $r_n = f_{n-1}(0)r_{n-1}/x$.

If $n = 1$ then $u_n = u_0 = 1$ so $2\deg(x^{n-1}u_n) = 0 = n-3+\delta_n$. Otherwise $2\deg(x^{n-2}u_{n-1}) \le n-4+\delta_{n-1}$ by the inductive hypothesis so $2\deg(x^{n-1}u_n) \le n-3+\delta_n$ as above.

Also $2\deg(x^{n-1}v_n) \le n-1+\delta_n$ as above; $2\deg(x^n q_n) = 2\deg(x^{n-1}q_{n-1}) \le n-2-\delta_{n-1} = n-1-\delta_n$; and $2\deg(x^n r_n) = 2\deg(x^{n-1}r_{n-1}) \le n-\delta_{n-1} = n+1-\delta_n$.

Case 3: $\delta_{n-1} > 0$ and $g_{n-1}(0) \ne 0$. Then $\delta_n = 1-\delta_{n-1}$; $u_n = q_{n-1}$; $v_n = r_{n-1}$; $q_n = (g_{n-1}(0)u_{n-1} - f_{n-1}(0)q_{n-1})/x$; and $r_n = (g_{n-1}(0)v_{n-1} - f_{n-1}(0)r_{n-1})/x$.

If $n = 1$ then $u_n = q_0 = 0$ so $2\deg(x^{n-1}u_n) = -\infty \le n-3+\delta_n$. Otherwise $2\deg(x^{n-1}q_{n-1}) \le n-2-\delta_{n-1}$ by the inductive hypothesis so $2\deg(x^{n-1}u_n) \le n-2-\delta_{n-1} = n-3+\delta_n$. Similarly $2\deg(x^{n-1}r_{n-1}) \le n-\delta_{n-1}$ by the inductive hypothesis so $2\deg(x^{n-1}v_n) \le n-\delta_{n-1} = n-1+\delta_n$.

Finally, for q_n, note that both $x^{n-1}u_{n-1}$ and $x^{n-1}q_{n-1}$ have degree at most $(n-2+\delta_{n-1})/2$, so the same is true for their k-linear combination $x^n q_n$, so $2\deg(x^n q_n) \le n-2+\delta_{n-1} = n-1-\delta_n$. Similarly $2\deg(x^n r_n) \le n+\delta_{n-1} = n+1-\delta_n$. □

First proof of Theorem 6.2. Write $\varphi = f_{2d-1}$. By Theorem A.1, $2\deg\varphi \le \delta_{2d-1}$ and $2\deg g_{2d-1} \le -\delta_{2d-1}$. Each f_n is nonzero by construction, so $\deg\varphi \ge 0$, so $\delta_{2d-1} \ge 0$.

By Theorem A.2, $x^{2d-2}u_{2d-1}$ is a polynomial of degree at most $d-2+\delta_{2d-1}/2$. Define $C_0 = x^{-d+\delta_{2d-1}/2}u_{2d-1}(1/x)$; then C_0 is a polynomial of degree at most $d-2+\delta_{2d-1}/2$. Similarly define $C_1 = x^{-d+1+\delta_{2d-1}/2}v_{2d-1}(1/x)$, and observe that C_1 is a polynomial of degree at most $d-1+\delta_{2d-1}/2$.

Multiply C_0 by $R_0 = x^d f(1/x)$, multiply C_1 by $R_1 = x^{d-1}g(1/x)$, and add, to obtain

$$\begin{aligned} C_0R_0 + C_1R_1 &= x^{\delta_{2d-1}/2}(u_{2d-1}(1/x)f(1/x) + v_{2d-1}(1/x)g(1/x)) \\ &= x^{\delta_{2d-1}/2}\varphi(1/x) \end{aligned}$$

by Theorem 4.2.

Define Γ as the monic polynomial $x^{\delta_{2d-1}/2}\varphi(1/x)/\varphi(0)$ of degree $\delta_{2d-1}/2$. We have just shown that $\Gamma \in R_0k[x] + R_1k[x]$: specifically, $\Gamma = (C_0/\varphi(0))R_0 + (C_1/\varphi(0))R_1$. To finish the proof we will show that $R_0 \in \Gamma k[x]$. This forces $\Gamma = \gcd\{R_0, R_1\} = G$, which in turn implies $\deg G = \delta_{2d-1}/2$ and $V = C_1/\varphi(0)$.

Observe that $\delta_{2d} = 1+\delta_{2d-1}$ and $f_{2d} = \varphi$; the "swapped" case is impossible here since $\delta_{2d-1} > 0$ implies $g_{2d-1} = 0$. By Theorem A.1 again, $2\deg g_{2d} \le -1-\delta_{2d} = -2-\delta_{2d-1}$, so $g_{2d} = 0$.

By Theorem 4.2, $\varphi = f_{2d} = u_{2d}f + v_{2d}g$ and $0 = g_{2d} = q_{2d}f + r_{2d}g$, so $x^{2d}r_{2d}\varphi = \Delta f$ where $\Delta = x^{2d}(u_{2d}r_{2d} - v_{2d}q_{2d})$. By Theorem A.2, $\Delta \in k^*$, and $p = x^{2d}r_{2d}$ is a polynomial of degree at most $(2d+1-\delta_{2d})/2 = d-\delta_{2d-1}/2$. The polynomials $x^{d-\delta_{2d-1}/2}p(1/x)$ and $x^{\delta_{2d-1}/2}\varphi(1/x) = \Gamma$ have product $\Delta x^d f(1/x) = \Delta R_0$, so Γ divides R_0 as claimed. □

B Review of the Euclid–Stevin algorithm

This appendix reviews the Euclid–Stevin algorithm to compute polynomial gcd, including the extended version of the algorithm that writes each remainder as a linear combination of the two inputs.

Theorem B.1 (the Euclid–Stevin algorithm)**.** *Let k be a field. Let R_0, R_1 be elements of the polynomial ring $k[x]$. Recursively define a finite sequence of polynomials $R_2, R_3, \ldots, R_r \in k[x]$ as follows: if $i \geq 1$ and $R_i = 0$ then $r = i$; if $i \geq 1$ and $R_i \neq 0$ then $R_{i+1} = R_{i-1} \bmod R_i$. Define $d_i = \deg R_i$ and $\ell_i = R_i[d_i]$. Then*

- *$d_1 > d_2 > \cdots > d_r = -\infty$;*
- *$\ell_i \neq 0$ if $1 \leq i \leq r-1$;*
- *if $\ell_{r-1} \neq 0$ then $\gcd\{R_0, R_1\} = R_{r-1}/\ell_{r-1}$; and*
- *if $\ell_{r-1} = 0$ then $R_0 = R_1 = \gcd\{R_0, R_1\} = 0$ and $r = 1$.*

Proof. If $1 \leq i \leq r-1$ then $R_i \neq 0$ so $\ell_i = R_i[\deg R_i] \neq 0$. Also $R_{i+1} = R_{i-1} \bmod R_i$ so $\deg R_{i+1} < \deg R_i$, i.e., $d_{i+1} < d_i$. Hence $d_1 > d_2 > \cdots > d_r$; and $d_r = \deg R_r = \deg 0 = -\infty$.

If $\ell_{r-1} = 0$ then r must be 1, so $R_1 = 0$ (since $R_r = 0$) and $R_0 = 0$ (since $\ell_0 = 0$), so $\gcd\{R_0, R_1\} = 0$. Assume from now on that $\ell_{r-1} \neq 0$. The divisibility of $R_{i+1} - R_{i-1}$ by R_i implies $\gcd\{R_{i-1}, R_i\} = \gcd\{R_i, R_{i+1}\}$, so $\gcd\{R_0, R_1\} = \gcd\{R_1, R_2\} = \cdots = \gcd\{R_{r-1}, R_r\} = \gcd\{R_{r-1}, 0\} = R_{r-1}/\ell_{r-1}$. □

Theorem B.2 (the extended Euclid–Stevin algorithm)**.** *In the situation of Theorem B.1, define $U_0, V_0, \ldots, U_r, U_r \in k[x]$ as follows: $U_0 = 1$; $V_0 = 0$; $U_1 = 0$; $V_1 = 1$; $U_{i+1} = U_{i-1} - \lfloor R_{i-1}/R_i \rfloor U_i$ for $i \in \{1, \ldots, r-1\}$; $V_{i+1} = V_{i-1} - \lfloor R_{i-1}/R_i \rfloor V_i$ for $i \in \{1, \ldots, r-1\}$. Then*

- *$R_i = U_i R_0 + V_i R_1$ for $i \in \{0, \ldots, r\}$;*
- *$U_i V_{i+1} - U_{i+1} V_i = (-1)^i$ for $i \in \{0, \ldots, r-1\}$;*
- *$V_{i+1} R_i - V_i R_{i+1} = (-1)^i R_0$ for $i \in \{0, \ldots, r-1\}$; and*
- *if $d_0 > d_1$ then $\deg V_{i+1} = d_0 - d_i$ for $i \in \{0, \ldots, r-1\}$.*

Proof. $U_0 R_0 + V_0 R_1 = 1R_0 + 0R_1 = R_0$ and $U_1 R_0 + V_1 R_1 = 0R_0 + 1R_1 = R_1$. For $i \in \{1, \ldots, r-1\}$, assume inductively that $R_{i-1} = U_{i-1} R_0 + V_{i-1} R_1$ and $R_i = U_i R_0 + V_i R_1$, and write $Q_i = \lfloor R_{i-1}/R_i \rfloor$. Then $R_{i+1} = R_{i-1} \bmod R_i = R_{i-1} - Q_i R_i = (U_{i-1} - Q_i U_i) R_0 + (V_{i-1} - Q_i V_i) R_1 = U_{i+1} R_0 + V_{i+1} R_1$.

If $i = 0$ then $U_i V_{i+1} - U_{i+1} V_i = 1 \cdot 1 - 0 \cdot 0 = 1 = (-1)^i$. For $i \in \{1, \ldots, r-1\}$, assume inductively that $U_{i-1} V_i - U_i V_{i-1} = (-1)^{i-1}$; then $U_i V_{i+1} - U_{i+1} V_i = U_i (V_{i-1} - Q V_i) - (U_{i-1} - Q U_i) V_i = -(U_{i-1} V_i - U_i V_{i-1}) = (-1)^i$.

Multiply $R_i = U_i R_0 + V_i R_1$ by V_{i+1}, multiply $R_{i+1} = U_{i+1} R_0 + V_{i+1} R_1$ by U_{i+1}, and subtract, to see that $V_{i+1} R_i - V_i R_{i+1} = (-1)^i R_0$.

Finally, assume $d_0 > d_1$. If $i = 0$ then $\deg V_{i+1} = \deg 1 = 0 = d_0 - d_i$. For $i \in \{1, \ldots, r-1\}$, assume inductively that $\deg V_i = d_0 - d_{i-1}$. Then $\deg V_i R_{i+1} < d_0$ since $\deg R_{i+1} = d_{i+1} < d_{i-1}$; so $\deg V_{i+1} R_i = \deg(V_i R_{i+1} + (-1)^i R_0) = d_0$, so $\deg V_{i+1} = d_0 - d_i$. □

C Euclid–Stevin results as iterates of divstep

If the Euclid–Stevin algorithm is written as a sequence of polynomial divisions, and each polynomial division is written as a sequence of coefficient-elimination steps, then each coefficient-elimination step corresponds to one application of divstep. The exact correspondence is stated in Theorem C.2. This generalizes the known Berlekamp–Massey equivalence mentioned in Section 1.

Theorem C.1 is a warmup, showing how a single polynomial division relates to a sequence of division steps.

Theorem C.1 (polynomial division as a sequence of division steps)**.** *Let k be a field. Let R_0, R_1 be elements of the polynomial ring $k[x]$. Write $d_0 = \deg R_0$ and $d_1 = \deg R_1$. Assume that $d_0 > d_1 \geq 0$. Then*

$$\begin{aligned}&\text{divstep}^{2d_0-2d_1}(1, x^{d_0}R_0(1/x), x^{d_0-1}R_1(1/x))\\ &\qquad = (1, \ell_0^{d_0-d_1-1}x^{d_1}R_1(1/x), (\ell_0^{d_0-d_1-1}\ell_1)^{d_0-d_1+1}x^{d_1-1}R_2(1/x))\end{aligned}$$

where $\ell_0 = R_0[d_0]$, $\ell_1 = R_1[d_1]$, and $R_2 = R_0 \bmod R_1$.

Proof. **Part 1: The first $d_0 - d_1 - 1$ iterations.** If $\delta \in \{1, 2, \ldots, d_0 - d_1 - 1\}$ then $d_0 - \delta > d_1$, so $R_1[d_0 - \delta] = 0$, so the polynomial $\ell_0^{\delta-1}x^{d_0-\delta}R_1(1/x)$ has constant coefficient 0. The polynomial $x^{d_0}R_0(1/x)$ has constant coefficient ℓ_0. Hence

$$\text{divstep}(\delta, x^{d_0}R_0(1/x), \ell_0^{\delta-1}x^{d_0-\delta}R_1(1/x)) = (1+\delta, x^{d_0}R_0(1/x), \ell_0^{\delta}x^{d_0-\delta-1}R_1(1/x))$$

by definition of divstep. By induction

$$\begin{aligned}&\text{divstep}^{d_0-d_1-1}(1, x^{d_0}R_0(1/x), x^{d_0-1}R_1(1/x))\\ &\qquad = (d_0-d_1, x^{d_0}R_0(1/x), \ell_0^{d_0-d_1-1}x^{d_1}R_1(1/x))\\ &\qquad = (d_0-d_1, x^{d_0}R_0(1/x), x^{d_1}R_1'(1/x))\end{aligned}$$

where $R_1' = \ell_0^{d_0-d_1-1}R_1$. Note that $R_1'[d_1] = \ell_1'$ where $\ell_1' = \ell_0^{d_0-d_1-1}\ell_1$.

Part 2: The swap iteration, and the last $d_0 - d_1$ iterations. Define

$$\begin{aligned}Z_{d_0-d_1} &= R_0 \bmod x^{d_0-d_1}R_1';\\ Z_{d_0-d_1+1} &= R_0 \bmod x^{d_0-d_1-1}R_1';\\ &\vdots\\ Z_{2d_0-2d_1} &= R_0 \bmod R_1' = R_0 \bmod R_1 = R_2.\end{aligned}$$

Then

$$\begin{aligned}Z_{d_0-d_1} &= R_0 - (R_0[d_0]/\ell_1')x^{d_0-d_1}R_1';\\ Z_{d_0-d_1+1} &= Z_{d_0-d_1} - (Z_{d_0-d_1}[d_0-1]/\ell_1')x^{d_0-d_1-1}R_1';\\ &\vdots\\ Z_{2d_0-2d_1} &= Z_{2d_0-2d_1-1} - (Z_{2d_0-2d_1-1}[d_1]/\ell_1')R_1'.\end{aligned}$$

Substitute $1/x$ for x to see that

$$\begin{aligned}x^{d_0}Z_{d_0-d_1}(1/x) &= x^{d_0}R_0(1/x) - (R_0[d_0]/\ell_1')x^{d_1}R_1'(1/x);\\ x^{d_0-1}Z_{d_0-d_1+1}(1/x) &= x^{d_0-1}Z_{d_0-d_1}(1/x) - (Z_{d_0-d_1}[d_0-1]/\ell_1')x^{d_1}R_1'(1/x);\\ &\vdots\\ x^{d_1}Z_{2d_0-2d_1}(1/x) &= x^{d_1}Z_{2d_0-2d_1-1}(1/x) - (Z_{2d_0-2d_1-1}[d_1]/\ell_1')x^{d_1}R_1'(1/x).\end{aligned}$$

We will use these equations to show that the $d_0 - d_1, \ldots, 2d_0 - 2d_1$ iterates of divstep compute $\ell_1'x^{d_0-1}Z_{d_0-d_1}, \ldots, (\ell_1')^{d_0-d_1+1}x^{d_1-1}Z_{2d_0-2d_1}$ respectively.

The constant coefficients of $x^{d_0}R_0(1/x)$ and $x^{d_1}R_1'(1/x)$ are $R_0[d_0]$ and $\ell_1' \neq 0$ respectively, and $\ell_1'x^{d_0}R_0(1/x) - R_0[d_0]x^{d_1}R_1'(1/x) = \ell_1'x^{d_0}Z_{d_0-d_1}(1/x)$, so

$$\begin{aligned}&\text{divstep}(d_0-d_1, x^{d_0}R_0(1/x), x^{d_1}R_1'(1/x))\\ &\qquad = (1-(d_0-d_1), x^{d_1}R_1'(1/x), \ell_1'x^{d_0-1}Z_{d_0-d_1}(1/x)).\end{aligned}$$

The constant coefficients of $x^{d_1}R_1'(1/x)$ and $\ell_1' x^{d_0-1}Z_{d_0-d_1}(1/x)$ are ℓ_1' and $\ell_1' Z_{d_0-d_1}[d_0-1]$ respectively, and

$$(\ell_1')^2 x^{d_0-1}Z_{d_0-d_1}(1/x) - \ell_1' Z_{d_0-d_1}[d_0-1]x^{d_1}R_1'(1/x) = (\ell_1')^2 x^{d_0-1}Z_{d_0-d_1+1}(1/x),$$

so

$$\begin{aligned}&\operatorname{divstep}(1-(d_0-d_1), x^{d_1}R_1'(1/x), \ell_1' x^{d_0-1}Z_{d_0-d_1}(1/x))\\ &\quad = (2-(d_0-d_1), x^{d_1}R_1'(1/x), (\ell_1')^2 x^{d_0-2}Z_{d_0-d_1+1}(1/x)).\end{aligned}$$

Continue in this way to see that

$$\begin{aligned}&\operatorname{divstep}^i(d_0-d_1, x^{d_0}R_0(1/x), x^{d_1}R_1'(1/x))\\ &\quad = (i-(d_0-d_1), x^{d_1}R_1'(1/x), (\ell_1')^i x^{d_0-i}Z_{d_0-d_1+i-1}(1/x))\end{aligned}$$

for $1 \le i \le d_0 - d_1 + 1$. In particular,

$$\begin{aligned}&\operatorname{divstep}^{2d_0-2d_1}(1, x^{d_0}R_0(1/x), x^{d_0-1}R_1(1/x))\\ &= \operatorname{divstep}^{d_0-d_1+1}(d_0-d_1, x^{d_0}R_0(1/x), x^{d_1}R_1'(1/x))\\ &\quad = (1, x^{d_1}R_1'(1/x), (\ell_1')^{d_0-d_1+1}x^{d_1-1}R_2(1/x))\\ &\quad = (1, \ell_0^{d_0-d_1-1}x^{d_1}R_1(1/x), (\ell_0^{d_0-d_1-1}\ell_1)^{d_0-d_1+1}x^{d_1-1}R_2(1/x))\end{aligned}$$

as claimed. □

Theorem C.2. *In the situation of Theorem B.2, assume that $d_0 > d_1$. Define $f = x^{d_0}R_0(1/x)$ and $g = x^{d_0-1}R_1(1/x)$. Then $f, g \in k[x]$. Define $(\delta_n, f_n, g_n) = \operatorname{divstep}^n(1, f, g)$ and $\mathcal{T}_n = \mathcal{T}(\delta_n, f_n, g_n)$.*

Part A. *If $i \in \{0, 1, \ldots, r-2\}$ and $2d_0 - 2d_i \le n < 2d_0 - d_i - d_{i+1}$ then*

$$\begin{aligned}\delta_n &= 1 + n - (2d_0 - 2d_i) > 0,\\ f_n &= a_n x^{d_i} R_i(1/x),\\ g_n &= b_n x^{2d_0-d_i-n-1} R_{i+1}(1/x),\\ \mathcal{T}_{n-1}\cdots\mathcal{T}_0 &= \begin{pmatrix} a_n(1/x)^{d_0-d_i}U_i(1/x) & a_n(1/x)^{d_0-d_i-1}V_i(1/x)\\ b_n(1/x)^{n-(d_0-d_i)+1}U_{i+1}(1/x) & b_n(1/x)^{n-(d_0-d_i)}V_{i+1}(1/x)\end{pmatrix}\end{aligned}$$

for some $a_n, b_n \in k^*$.

Part B. *If $i \in \{0, 1, \ldots, r-2\}$ and $2d_0 - d_i - d_{i+1} \le n < 2d_0 - 2d_{i+1}$ then*

$$\begin{aligned}\delta_n &= 1 + n - (2d_0 - 2d_{i+1}) \le 0,\\ f_n &= a_n x^{d_{i+1}} R_{i+1}(1/x),\\ g_n &= b_n x^{2d_0-d_{i+1}-n-1} Z_n(1/x),\\ \mathcal{T}_{n-1}\cdots\mathcal{T}_0 &= \begin{pmatrix} a_n(1/x)^{d_0-d_{i+1}}U_{i+1}(1/x) & a_n(1/x)^{d_0-d_{i+1}-1}V_{i+1}(1/x)\\ b_n(1/x)^{n-(d_0-d_{i+1})+1}X_n(1/x) & b_n(1/x)^{n-(d_0-d_{i+1})}Y_n(1/x)\end{pmatrix}\end{aligned}$$

for some $a_n, b_n \in k^*$, where

$$\begin{aligned}Z_n &= R_i \bmod x^{2d_0-2d_{i+1}-n}R_{i+1},\\ X_n &= U_i - \left\lfloor R_i/x^{2d_0-2d_{i+1}-n}R_{i+1}\right\rfloor x^{2d_0-2d_{i+1}-n}U_{i+1},\\ Y_n &= V_i - \left\lfloor R_i/x^{2d_0-2d_{i+1}-n}R_{i+1}\right\rfloor x^{2d_0-2d_{i+1}-n}V_{i+1}.\end{aligned}$$

Part C. *If* $n \geq 2d_0 - 2d_{r-1}$ *then*

$$\begin{aligned}\delta_n &= 1 + n - (2d_0 - 2d_{r-1}) > 0,\\ f_n &= a_n x^{d_{r-1}} R_{r-1}(1/x),\\ g_n &= 0,\\ \mathcal{T}_{n-1} \cdots \mathcal{T}_0 &= \begin{pmatrix} a_n(1/x)^{d_0-d_{r-1}} U_{r-1}(1/x) & a_n(1/x)^{d_0-d_{r-1}-1} V_{r-1}(1/x) \\ b_n(1/x)^{n-(d_0-d_{r-1})+1} U_r(1/x) & b_n(1/x)^{n-(d_0-d_{r-1})} V_r(1/x) \end{pmatrix}\end{aligned}$$

for some $a_n, b_n \in k^*$.

The proof also gives recursive formulas for a_n, b_n: namely, $(a_0, b_0) = (1, 1)$ for the base case, $(a_n, b_n) = (b_{n-1}, g_{n-1}(0)a_{n-1})$ for the swapped case, and $(a_n, b_n) = (a_{n-1}, f_{n-1}(0)b_{n-1})$ for the unswapped case. One can, if desired, augment divstep to track a_n and b_n; these are the denominators eliminated in a "fraction-free" computation.

Proof. By hypothesis $\deg R_0 = d_0 > d_1 = \deg R_1$. Hence d_0 is a nonnegative integer, and $f = R_0[d_0] + R_0[d_0 - 1]x + \cdots + R_0[0]x^{d_0} \in k[x]$. Similarly $g = R_1[d_0 - 1] + R_1[d_0 - 2]x + \cdots + R_1[0]x^{d_0-1} \in k[x]$; this is also true for $d_0 = 0$, with $R_1 = 0$ and $g = 0$.

We induct on n, and split the analysis of n into six cases. There are three different formulas (A, B, C) applicable to different ranges of n, and within each range we consider the first n separately. For brevity we write $\overline{R}_i = R_i(1/x)$, $\overline{U}_i = U_i(1/x)$, $\overline{V}_i = V_i(1/x)$, $\overline{X}_n = X_n(1/x)$, $\overline{Y}_n = Y_n(1/x)$, $\overline{Z}_n = Z_n(1/x)$.

Case A1: $n = 2d_0 - 2d_i$ for some $i \in \{0, 1, \ldots, r-2\}$.

If $n = 0$ then $i = 0$. By assumption $\delta_n = 1$ as claimed; $f_n = f = x^{d_0}\overline{R}_0$ so $f_n = a_n x^{d_0}\overline{R}_0$ as claimed where $a_n = 1$; $g_n = g = x^{d_0-1}\overline{R}_1$ so $g_n = b_n x^{d_0-1}\overline{R}_1$ as claimed where $b_n = 1$. Also $U_i = 1$, $U_{i+1} = 0$, $V_i = 0$, $V_{i+1} = 1$, and $d_0 - d_i = 0$, so

$$\begin{pmatrix} a_n(1/x)^{d_0-d_i}\overline{U}_i & a_n(1/x)^{d_0-d_i-1}\overline{V}_i \\ b_n(1/x)^{d_0-d_i+1}\overline{U}_{i+1} & b_n(1/x)^{d_0-d_i}\overline{V}_{i+1} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \mathcal{T}_{n-1} \cdots \mathcal{T}_0$$

as claimed.

Otherwise $n > 0$ so $i > 0$. Now $2d_0 - d_{i-1} - d_i \leq n - 1 < 2d_0 - 2d_i$ so, by the inductive hypothesis, $\delta_{n-1} = 1 + (n-1) - (2d_0 - 2d_i) = 0$; $f_{n-1} = a_{n-1}x^{d_i}\overline{R}_i$ for some $a_{n-1} \in k^*$; $g_{n-1} = b_{n-1}x^{d_i}\overline{Z}_{n-1}$ for some $b_{n-1} \in k^*$, where $Z_{n-1} = R_{i-1} \bmod xR_i$; and

$$\mathcal{T}_{n-2} \cdots \mathcal{T}_0 = \begin{pmatrix} a_{n-1}(1/x)^{d_0-d_i}\overline{U}_i & a_{n-1}(1/x)^{d_0-d_i-1}\overline{V}_i \\ b_{n-1}(1/x)^{d_0-d_i}\overline{X}_{n-1} & b_{n-1}(1/x)^{d_0-d_i-1}\overline{Y}_{n-1} \end{pmatrix}$$

where $X_n = U_{i-1} - \lfloor R_{i-1}/xR_i \rfloor xU_i$ and $Y_n = V_{i-1} - \lfloor R_{i-1}/xR_i \rfloor xV_i$.

Note that $\deg Z_{n-1} \leq d_i$. Observe that

$$\begin{aligned} R_{i+1} = R_{i-1} \bmod R_i &= Z_{n-1} - (Z_{n-1}[d_i]/\ell_i)R_i,\\ U_{i+1} &= X_{n-1} - (Z_{n-1}[d_i]/\ell_i)U_i,\\ V_{i+1} &= Y_{n-1} - (Z_{n-1}[d_i]/\ell_i)V_i.\end{aligned}$$

The point is that subtracting $(Z_{n-1}[d_i]/\ell_i)R_i$ from Z_{n-1} eliminates the coefficient of x^{d_i} and produces a polynomial of degree at most $d_i - 1$, namely $Z_{n-1} \bmod R_i = R_{i-1} \bmod R_i$.

Starting from $R_{i+1} = Z_{n-1} - (Z_{n-1}[d_i]/\ell_i)R_i$, substitute $1/x$ for x and multiply by $a_{n-1}b_{n-1}\ell_i x^{d_i}$ to see that

$$\begin{aligned} a_{n-1}b_{n-1}\ell_i x^{d_i}\overline{R}_{i+1} &= a_{n-1}\ell_i g_{n-1} - b_{n-1}Z_{n-1}[d_i]f_{n-1}\\ &= f_{n-1}(0)g_{n-1} - g_{n-1}(0)f_{n-1}.\end{aligned}$$

Now

$$\begin{aligned}(\delta_n, f_n, g_n) &= \operatorname{divstep}(\delta_{n-1}, f_{n-1}, g_{n-1}) \\ &= (1+\delta_{n-1}, f_{n-1}, (f_{n-1}(0)g_{n-1} - g_{n-1}(0)f_{n-1})/x).\end{aligned}$$

Hence $\delta_n = 1 = 1 + n - (2d_0 - 2d_i) > 0$ as claimed; $f_n = a_n x^{d_i}\overline{R}_i$ where $a_n = a_{n-1} \in k^*$ as claimed; and $g_n = b_n x^{d_i-1}\overline{R}_{i+1}$ where $b_n = a_{n-1}b_{n-1}\ell_i \in k^*$ as claimed.

Finally, $\overline{U}_{i+1} = \overline{X}_{n-1} - (Z_{n-1}[d_i]/\ell_i)\overline{U}_i$ and $\overline{V}_{i+1} = \overline{Y}_{n-1} - (Z_{n-1}[d_i]/\ell_i)\overline{V}_i$. Substitute these into the matrix

$$\begin{pmatrix} a_n(1/x)^{d_0-d_i}\overline{U}_i & a_n(1/x)^{d_0-d_i-1}\overline{V}_i \\ b_n(1/x)^{d_0-d_i+1}\overline{U}_{i+1} & b_n(1/x)^{d_0-d_i}\overline{V}_{i+1} \end{pmatrix},$$

along with $a_n = a_{n-1}$ and $b_n = a_{n-1}b_{n-1}\ell_i$, to see that this matrix is

$$\mathcal{T}_{n-1} = \begin{pmatrix} 1 & 0 \\ -b_{n-1}Z_{n-1}[d_i]/x & a_{n-1}\ell_i/x \end{pmatrix}$$

times $\mathcal{T}_{n-2}\cdots\mathcal{T}_0$ shown above.

Case A2: $2d_0 - 2d_i < n < 2d_0 - d_i - d_{i+1}$ for some $i \in \{0, 1, \ldots, r-2\}$.

By the inductive hypothesis, $\delta_{n-1} = n - (2d_0 - 2d_i) > 0$; $f_{n-1} = a_{n-1}x^{d_i}\overline{R}_i$ where $a_{n-1} \in k^*$; $g_{n-1} = b_{n-1}x^{2d_0-d_i-n}\overline{R}_{i+1}$ where $b_{n-1} \in k^*$; and

$$\mathcal{T}_{n-2}\cdots\mathcal{T}_0 = \begin{pmatrix} a_{n-1}(1/x)^{d_0-d_i}\overline{U}_i & a_{n-1}(1/x)^{d_0-d_i-1}\overline{V}_i \\ b_{n-1}(1/x)^{n-(d_0-d_i)}\overline{U}_{i+1} & b_{n-1}(1/x)^{n-(d_0-d_i)-1}\overline{V}_{i+1} \end{pmatrix}.$$

Note that $g_{n-1}(0) = b_{n-1}R_{i+1}[2d_0 - d_i - n] = 0$ since $2d_0 - d_i - n > d_{i+1} = \deg R_{i+1}$. Thus

$$(\delta_n, f_n, g_n) = \operatorname{divstep}(\delta_{n-1}, f_{n-1}, g_{n-1}) = (1+\delta_{n-1}, f_{n-1}, f_{n-1}(0)g_{n-1}/x).$$

Hence $\delta_n = 1 + n - (2d_0 - 2d_i) > 0$ as claimed; $f_n = a_n x^{d_i}\overline{R}_i$ where $a_n = a_{n-1} \in k^*$ as claimed; and $g_n = b_n x^{2d_0-d_i-n-1}\overline{R}_{i+1}$ where $b_n = f_{n-1}(0)b_{n-1} = a_{n-1}b_{n-1}\ell_i \in k^*$ as claimed.

Also $\mathcal{T}_{n-1} = \begin{pmatrix} 1 & 0 \\ 0 & a_{n-1}\ell_i/x \end{pmatrix}$. Multiply by $\mathcal{T}_{n-2}\cdots\mathcal{T}_0$ above to see that

$$\mathcal{T}_{n-1}\cdots\mathcal{T}_0 = \begin{pmatrix} a_n(1/x)^{d_0-d_i}\overline{U}_i & a_n(1/x)^{d_0-d_i-1}\overline{V}_i \\ b_n(1/x)^{n-(d_0-d_i)+1}\overline{U}_{i+1} & b_n\ell_i(1/x)^{n-(d_0-d_i)}\overline{V}_{i+1} \end{pmatrix}$$

as claimed.

Case B1: $n = 2d_0 - d_i - d_{i+1}$ for some $i \in \{0, 1, \ldots, r-2\}$.

Note that $2d_0 - 2d_i \le n - 1 < 2d_0 - d_i - d_{i+1}$. By the inductive hypothesis, $\delta_{n-1} = d_i - d_{i+1} > 0$; $f_{n-1} = a_{n-1}x^{d_i}\overline{R}_i$ where $a_{n-1} \in k^*$; $g_{n-1} = b_{n-1}x^{d_{i+1}}\overline{R}_{i+1}$ where $b_{n-1} \in k^*$; and

$$\mathcal{T}_{n-2}\cdots\mathcal{T}_0 = \begin{pmatrix} a_{n-1}(1/x)^{d_0-d_i}\overline{U}_i & a_{n-1}(1/x)^{d_0-d_i-1}\overline{V}_i \\ b_{n-1}(1/x)^{d_0-d_{i+1}}\overline{U}_{i+1} & b_{n-1}\ell_i(1/x)^{d_0-d_{i+1}-1}\overline{V}_{i+1} \end{pmatrix}.$$

Observe that

$$\begin{aligned} Z_n &= R_i - (\ell_i/\ell_{i+1})x^{d_i-d_{i+1}}R_{i+1}, \\ X_n &= U_i - (\ell_i/\ell_{i+1})x^{d_i-d_{i+1}}U_{i+1}, \\ Y_n &= V_i - (\ell_i/\ell_{i+1})x^{d_i-d_{i+1}}V_{i+1}. \end{aligned}$$

Substitute $1/x$ for x in the Z_n equation and multiply by $a_{n-1}b_{n-1}\ell_{i+1}x^{d_i}$ to see that

$$\begin{aligned} a_{n-1}b_{n-1}\ell_{i+1}x^{d_i}\overline{Z}_n &= b_{n-1}\ell_{i+1}f_{n-1} - a_{n-1}\ell_i g_{n-1} \\ &= g_{n-1}(0)f_{n-1} - f_{n-1}(0)g_{n-1}. \end{aligned}$$

Also note that $g_{n-1}(0) = b_{n-1}\ell_{i+1} \neq 0$ so

$$\begin{aligned}(\delta_n, f_n, g_n) &= \operatorname{divstep}(\delta_{n-1}, f_{n-1}, g_{n-1}) \\ &= (1-\delta_{n-1}, g_{n-1}, (g_{n-1}(0)f_{n-1} - f_{n-1}(0)g_{n-1})/x).\end{aligned}$$

Hence $\delta_n = 1-(d_i-d_{i+1}) \leq 0$ as claimed; $f_n = a_n x^{d_{i+1}}\overline{R}_{i+1}$ where $a_n = b_{n-1} \in k^*$ as claimed; and $g_n = b_n x^{d_i-1}\overline{Z}_n$ where $b_n = a_{n-1}b_{n-1}\ell_{i+1} \in k^*$ as claimed.

Finally, substitute $\overline{X}_n = \overline{U}_i - (\ell_i/\ell_{i+1})(1/x)^{d_i-d_{i+1}}\overline{U}_{i+1}$ and substitute $\overline{Y}_n = \overline{V}_i - (\ell_i/\ell_{i+1})(1/x)^{d_i-d_{i+1}}\overline{V}_{i+1}$ into the matrix

$$\begin{pmatrix} a_n(1/x)^{d_0-d_{i+1}}\overline{U}_{i+1} & a_n(1/x)^{d_0-d_{i+1}-1}\overline{V}_{i+1} \\ b_n(1/x)^{d_0-d_i+1}\overline{X}_n & b_n(1/x)^{d_0-d_i}\overline{Y}_n \end{pmatrix},$$

along with $a_n = b_{n-1}$ and $b_n = a_{n-1}b_{n-1}\ell_{i+1}$, to see that this matrix is $\mathcal{T}_{n-1} = \begin{pmatrix} 0 & 1 \\ b_{n-1}\ell_{i+1}/x & -a_{n-1}\ell_i/x \end{pmatrix}$ times $\mathcal{T}_{n-2}\cdots\mathcal{T}_0$ shown above.

Case B2: $2d_0 - d_i - d_{i+1} < n < 2d_0 - 2d_{i+1}$ for some $i \in \{0, 1, \ldots, r-2\}$.

Now $2d_0-d_i-d_{i+1} \leq n-1 < 2d_0-2d_{i+1}$. By the inductive hypothesis, $\delta_{n-1} = n-(2d_0-2d_{i+1}) \leq 0$; $f_{n-1} = a_{n-1}x^{d_{i+1}}\overline{R}_{i+1}$ for some $a_{n-1} \in k^*$; $g_{n-1} = b_{n-1}x^{2d_0-d_{i+1}-n}\overline{Z}_{n-1}$ for some $b_{n-1} \in k^*$, where $Z_{n-1} = R_i \bmod x^{2d_0-2d_{i+1}-n+1}R_{i+1}$; and

$$\mathcal{T}_{n-2}\cdots\mathcal{T}_0 = \begin{pmatrix} a_{n-1}(1/x)^{d_0-d_{i+1}}\overline{U}_{i+1} & a_{n-1}(1/x)^{d_0-d_{i+1}-1}\overline{V}_{i+1} \\ b_{n-1}(1/x)^{n-(d_0-d_{i+1})}\overline{X}_{n-1} & b_{n-1}(1/x)^{n-(d_0-d_{i+1})-1}\overline{Y}_{n-1} \end{pmatrix}$$

where $X_{n-1} = U_i - \lfloor R_i/x^{2d_0-2d_{i+1}-n+1}R_{i+1}\rfloor x^{2d_0-2d_{i+1}-n+1}U_{i+1}$ and $Y_{n-1} = V_i - \lfloor R_i/x^{2d_0-2d_{i+1}-n+1}R_{i+1}\rfloor x^{2d_0-2d_{i+1}-n+1}V_{i+1}$.

Observe that

$$\begin{aligned}Z_n &= R_i \bmod x^{2d_0-2d_{i+1}-n}R_{i+1} \\ &= Z_{n-1} - (Z_{n-1}[2d_0-d_{i+1}-n]/\ell_{i+1})x^{2d_0-2d_{i+1}-n}R_{i+1}, \\ X_n &= X_{n-1} - (Z_{n-1}[2d_0-d_{i+1}-n]/\ell_{i+1})x^{2d_0-2d_{i+1}-n}U_{i+1}, \\ Y_n &= Y_{n-1} - (Z_{n-1}[2d_0-d_{i+1}-n]/\ell_{i+1})x^{2d_0-2d_{i+1}-n}V_{i+1}.\end{aligned}$$

The point is that $\deg Z_{n-1} \leq 2d_0-d_{i+1}-n$, and subtracting

$$(Z_{n-1}[2d_0-d_{i+1}-n]/\ell_{i+1})x^{2d_0-2d_{i+1}-n}R_{i+1}$$

from Z_{n-1} eliminates the coefficient of $x^{2d_0-d_{i+1}-n}$.

Starting from

$$Z_n = Z_{n-1} - (Z_{n-1}[2d_0-d_{i+1}-n]/\ell_{i+1})x^{2d_0-2d_{i+1}-n}R_{i+1}$$

substitute $1/x$ for x and multiply by $a_{n-1}b_{n-1}\ell_{i+1}x^{2d_0-d_{i+1}-n}$ to see that

$$\begin{aligned}a_{n-1}b_{n-1}\ell_{i+1}x^{2d_0-d_{i+1}-n}\overline{Z}_n &= a_{n-1}\ell_{i+1}g_{n-1} - b_{n-1}Z_{n-1}[2d_0-d_{i+1}-n]f_{n-1} \\ &= f_{n-1}(0)g_{n-1} - g_{n-1}(0)f_{n-1}.\end{aligned}$$

Now

$$\begin{aligned}(\delta_n, f_n, g_n) &= \operatorname{divstep}(\delta_{n-1}, f_{n-1}, g_{n-1}) \\ &= (1+\delta_{n-1}, f_{n-1}, (f_{n-1}(0)g_{n-1} - g_{n-1}(0)f_{n-1})/x).\end{aligned}$$

Hence $\delta_n = 1+n-(2d_0-2d_{i+1}) \leq 0$ as claimed; $f_n = a_n x^{d_{i+1}}\overline{R}_{i+1}$ where $a_n = a_{n-1} \in k^*$ as claimed; and $g_n = b_n x^{2d_0-d_{i+1}-n-1}\overline{Z}_n$ where $b_n = a_{n-1}b_{n-1}\ell_{i+1} \in k^*$ as claimed.

Finally, substitute

$$\overline{X}_n = \overline{X}_{n-1} - (Z_{n-1}[2d_0 - d_{i+1} - n]/\ell_{i+1})(1/x)^{2d_0-2d_{i+1}-n}\overline{U}_{i+1}$$

and

$$\overline{Y}_n = \overline{Y}_{n-1} - (Z_{n-1}[2d_0 - d_{i+1} - n]/\ell_{i+1})(1/x)^{2d_0-2d_{i+1}-n}\overline{V}_{i+1}$$

into the matrix

$$\begin{pmatrix} a_n(1/x)^{d_0-d_{i+1}}\overline{U}_{i+1} & a_n(1/x)^{d_0-d_{i+1}-1}\overline{V}_{i+1} \\ b_n(1/x)^{n-(d_0-d_{i+1})+1}\overline{X}_n & b_n(1/x)^{n-(d_0-d_{i+1})}\overline{Y}_n \end{pmatrix},$$

along with $a_n = a_{n-1}$ and $b_n = a_{n-1}b_{n-1}\ell_{i+1}$, to see that this matrix is $\mathcal{T}_{n-1} = \begin{pmatrix} 1 & 0 \\ -b_{n-1}Z_{n-1}[2d_0 - d_{i+1} - n]/x & a_{n-1}\ell_{i+1}/x \end{pmatrix}$ times $\mathcal{T}_{n-2}\cdots\mathcal{T}_0$ shown above.

Case C1: $n = 2d_0 - 2d_{r-1}$. This is essentially the same as Case A1, except that i is replaced with $r-1$, and g_n ends up as 0. For completeness we spell out the details.

If $n = 0$ then $r = 1$ and $R_1 = 0$ so $g = 0$. By assumption $\delta_n = 1$ as claimed; $f_n = f = x^{d_0}\overline{R}_0$ so $f_n = a_n x^{d_0}\overline{R}_0$ as claimed where $a_n = 1$; $g_n = g = 0$ as claimed. Define $b_n = 1$. Now $U_{r-1} = 1$, $U_r = 0$, $V_{r-1} = 0$, $V_r = 1$, and $d_0 - d_{r-1} = 0$, so

$$\begin{pmatrix} a_n(1/x)^{d_0-d_{r-1}}\overline{U}_{r-1} & a_n(1/x)^{d_0-d_{r-1}-1}\overline{V}_{r-1} \\ b_n(1/x)^{d_0-d_{r-1}+1}\overline{U}_r & b_n(1/x)^{d_0-d_{r-1}}\overline{V}_r \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \mathcal{T}_{n-1}\cdots\mathcal{T}_0$$

as claimed.

Otherwise $n > 0$ so $r \geq 2$. Now $2d_0 - d_{r-2} - d_{r-1} \leq n - 1 < 2d_0 - 2d_{r-1}$ so, by the inductive hypothesis (for $i = r-2$), $\delta_{n-1} = 1 + (n-1) - (2d_0 - 2d_{r-1}) = 0$; $f_{n-1} = a_{n-1}x^{d_{r-1}}\overline{R}_{r-1}$ for some $a_{n-1} \in k^*$; $g_{n-1} = b_{n-1}x^{d_{r-1}}\overline{Z}_{n-1}$ for some $b_{n-1} \in k^*$, where $Z_{n-1} = R_{r-2} \bmod xR_{r-1}$; and

$$\mathcal{T}_{n-2}\cdots\mathcal{T}_0 = \begin{pmatrix} a_{n-1}(1/x)^{d_0-d_{r-1}}\overline{U}_{r-1} & a_{n-1}(1/x)^{d_0-d_{r-1}-1}\overline{V}_{r-1} \\ b_{n-1}(1/x)^{d_0-d_{r-1}}\overline{X}_{n-1} & b_{n-1}(1/x)^{d_0-d_{r-1}-1}\overline{Y}_{n-1} \end{pmatrix}$$

where $X_n = U_{r-2} - \lfloor R_{r-2}/xR_{r-1}\rfloor xU_{r-1}$ and $Y_n = V_{r-2} - \lfloor R_{r-2}/xR_{r-1}\rfloor xV_{r-1}$.

Observe that

$$\begin{aligned} 0 = R_r = R_{r-2} \bmod R_{r-1} &= Z_{n-1} - (Z_{n-1}[d_{r-1}]/\ell_{r-1})R_{r-1}, \\ U_r &= X_{n-1} - (Z_{n-1}[d_{r-1}]/\ell_{r-1})U_{r-1}, \\ V_r &= Y_{n-1} - (Z_{n-1}[d_{r-1}]/\ell_{r-1})V_{r-1}. \end{aligned}$$

Starting from $Z_{n-1} - (Z_{n-1}[d_{r-1}]/\ell_{r-1})R_{r-1} = 0$, substitute $1/x$ for x and multiply by $a_{n-1}b_{n-1}\ell_{r-1}x^{d_{r-1}}$ to see that

$$a_{n-1}\ell_{r-1}g_{n-1} - b_{n-1}Z_{n-1}[d_{r-1}]f_{n-1} = 0,$$

i.e., $f_{n-1}(0)g_{n-1} - g_{n-1}(0)f_{n-1} = 0$. Now

$$(\delta_n, f_n, g_n) = \text{divstep}(\delta_{n-1}, f_{n-1}, g_{n-1}) = (1 + \delta_{n-1}, f_{n-1}, 0).$$

Hence $\delta_n = 1 = 1 + n - (2d_0 - 2d_{r-1}) > 0$ as claimed; $f_n = a_n x^{d_{r-1}}\overline{R}_{r-1}$ where $a_n = a_{n-1} \in k^*$ as claimed; and $g_n = 0$ as claimed.

Define $b_n = a_{n-1}b_{n-1}\ell_{r-1} \in k^*$, and substitute $\overline{U}_r = \overline{X}_{n-1} - (Z_{n-1}[d_{r-1}]/\ell_{r-1})\overline{U}_{r-1}$ and $\overline{V}_r = \overline{Y}_{n-1} - (Z_{n-1}[d_{r-1}]/\ell_{r-1})\overline{V}_{r-1}$ into the matrix

$$\begin{pmatrix} a_n(1/x)^{d_0-d_{r-1}}U_{r-1}(1/x) & a_n(1/x)^{d_0-d_{r-1}-1}V_{r-1}(1/x) \\ b_n(1/x)^{d_0-d_{r-1}+1}U_r(1/x) & b_n(1/x)^{d_0-d_{r-1}}V_r(1/x) \end{pmatrix},$$

to see that this matrix is $\mathcal{T}_{n-1} = \begin{pmatrix} 1 & 0 \\ -b_{n-1}Z_{n-1}[d_{r-1}]/x & a_{n-1}\ell_{r-1}/x \end{pmatrix}$ times $\mathcal{T}_{n-2}\cdots\mathcal{T}_0$ shown above.

Case C2: $n > 2d_0 - 2d_{r-1}$.

By the inductive hypothesis, $\delta_{n-1} = n - (2d_0 - 2d_{r-1})$; $f_{n-1} = a_{n-1}x^{d_{r-1}}\overline{R}_{r-1}$ for some $a_{n-1} \in k^*$; $g_{n-1} = 0$; and

$$\mathcal{T}_{n-2}\cdots\mathcal{T}_0 = \begin{pmatrix} a_{n-1}(1/x)^{d_0-d_{r-1}}U_{r-1}(1/x) & a_{n-1}(1/x)^{d_0-d_{r-1}-1}V_{r-1}(1/x) \\ b_{n-1}(1/x)^{n-(d_0-d_{r-1})}U_r(1/x) & b_{n-1}(1/x)^{n-(d_0-d_{r-1})-1}V_r(1/x) \end{pmatrix}$$

for some $b_{n-1} \in k^*$. Hence $\delta_n = 1 + \delta_n = 1 + n - (2d_0 - 2d_{r-1}) > 0$; $f_n = f_{n-1} = a_n x^{d_{r-1}}\overline{R}_{r-1}$ where $a_n = a_{n-1}$; and $g_n = 0$. Also $\mathcal{T}_{n-1} = \begin{pmatrix} 1 & 0 \\ 0 & a_{n-1}\ell_{r-1}/x \end{pmatrix}$ so

$$\mathcal{T}_{n-1}\cdots\mathcal{T}_0 = \begin{pmatrix} a_n(1/x)^{d_0-d_{r-1}}U_{r-1}(1/x) & a_n(1/x)^{d_0-d_{r-1}-1}V_{r-1}(1/x) \\ b_n(1/x)^{n-(d_0-d_{r-1})+1}U_r(1/x) & b_n(1/x)^{n-(d_0-d_{r-1})}V_r(1/x) \end{pmatrix}$$

where $b_n = a_{n-1}b_{n-1}\ell_{r-1} \in k^*$. □

D Alternative proof of main gcd theorem for polynomials

This appendix gives another proof of Theorem 6.2, using the relationship in Theorem C.2 between the Euclid–Stevin algorithm and iterates of divstep.

Second proof of Theorem 6.2. Define $R_2, R_3, \ldots, R_r$ and d_i and ℓ_i as in Theorem B.1; and define U_i and V_i as in Theorem B.2. Then $d_0 = d > d_1$, and all of the hypotheses of Theorems B.1, B.2, and C.2 are satisfied.

By Theorem B.1, $G = \gcd\{R_0, R_1\} = R_{r-1}/\ell_{r-1}$ (since $R_0 \neq 0$), so $\deg G = d_{r-1}$. Also, by Theorem B.2,

$$U_{r-1}R_0 + V_{r-1}R_1 = R_{r-1} = \ell_{r-1}G,$$

so $(V_{r-1}/\ell_{r-1})R_1 \equiv G \pmod{R_0}$; and $\deg V_{r-1} = d_0 - d_{r-2} < d_0 - d_{r-1} = d - \deg G$, so $V = V_{r-1}/\ell_{r-1}$.

Case 1: $d_{r-1} \geq 1$. Part C of Theorem C.2 applies to $n = 2d - 1$, since $n = 2d_0 - 1 \geq 2d_0 - 2d_{r-1}$. In particular, $\delta_n = 1 + n - (2d_0 - 2d_{r-1}) = 2d_{r-1} = 2\deg G$; $f_{2d-1} = a_{2d-1}x^{d_{r-1}}R_{r-1}(1/x)$; and $v_{2d-1} = a_{2d-1}(1/x)^{d_0-d_{r-1}-1}V_{r-1}(1/x)$.

Case 2: $d_{r-1} = 0$. Then $r \geq 2$ and $d_{r-2} \geq 1$. Part B of Theorem C.2 applies to $i = r - 2$ and $n = 2d - 1 = 2d_0 - 1$, since $2d_0 - d_i - d_{i+1} = 2d_0 - d_{r-2} \leq 2d_0 - 1 = n < 2d_0 = 2d_0 - 2d_{i+1}$. In particular, $\delta_n = 1 + n - (2d_0 - 2d_{i+1}) = 2d_{r-1} = 2\deg G$; again $f_{2d-1} = a_{2d-1}x^{d_{r-1}}R_{r-1}(1/x)$; and again $v_{2d-1} = a_{2d-1}(1/x)^{d_0-d_{r-1}-1}V_{r-1}(1/x)$.

In both cases, $f_{2d-1}(0) = a_{2d-1}\ell_{r-1}$, so $x^{\deg G}f_{2d-1}(1/x)/f_{2d-1}(0) = R_{r-1}/\ell_{r-1} = G$, and $x^{-d+1+\deg G}v_{2d-1}(1/x)/f_{2d-1}(0) = V_{r-1}/\ell_{r-1} = V$. □

E A gcd algorithm for integers

This appendix presents a variable-time algorithm to compute the gcd of two integers. This appendix also explains how a modification to the algorithm produces the Stehlé–Zimmermann algorithm [62].

Theorem E.1. *Let e be a positive integer. Let f be an element of $\mathbf{Z}_2^*$. Let g be an element of $2^e\mathbf{Z}_2^*$. Then there is a unique element $q \in \{1, 3, 5, \ldots, 2^{e+1} - 1\}$ such that $qg/2^e - f \in 2^{e+1}\mathbf{Z}_2$.*

We define $f \operatorname{div}_2 g = q/2^e \in \mathbf{Q}$, and we define $f \bmod_2 g = f - qg/2^e \in 2^{e+1}\mathbf{Z}_2$. Then $f = (f \operatorname{div}_2 g)g + (f \bmod_2 g)$. Note that $e = \operatorname{ord}_2 g$, so we are not creating any ambiguity by omitting e from the $f \operatorname{div}_2 g$ and $f \bmod_2 g$ notation.

Proof. First note that the quotient $f/(g/2^e)$ is odd. Define $q = (f/(g/2^e)) \bmod 2^{e+1}$; then $q \in \{1, 3, 5, \ldots, 2^{e+1}\}$, and $qg/2^e - f \in 2^{e+1}\mathbf{Z}_2$ by construction. To see uniqueness, note that if $q' \in \{1, 3, 5, \ldots, 2^{e+1} - 1\}$ also satisfies $q'g/2^e - f \in 2^{e+1}\mathbf{Z}_2$ then $(q - q')g/2^e \in 2^{e+1}\mathbf{Z}_2$, so $q - q' \in 2^{e+1}\mathbf{Z}_2$ since $g/2^e \in \mathbf{Z}_2^*$, so $q \bmod 2^{e+1} = q' \bmod 2^{e+1}$, so $q = q'$. □

Theorem E.2. *Let R_0 be a nonzero element of $\mathbf{Z}_2$. Let R_1 be an element of $2\mathbf{Z}_2$. Recursively define $e_0, e_1, e_2, e_3, \ldots \in \mathbf{Z}$ and $R_2, R_3, \ldots \in 2\mathbf{Z}_2$ as follows:*

- *If $i \geq 0$ and $R_i \neq 0$ then $e_i = \operatorname{ord}_2 R_i$.*
- *If $i \geq 1$ and $R_i \neq 0$ then $R_{i+1} = -((R_{i-1}/2^{e_{i-1}}) \bmod_2 R_i)/2^{e_i} \in 2\mathbf{Z}_2$.*
- *If $i \geq 1$ and $R_i = 0$ then $e_i, R_{i+1}, e_{i+1}, R_{i+2}, e_{i+2}, \ldots$ are undefined.*

Then

$$\begin{pmatrix} R_i/2^{e_i} \\ R_{i+1} \end{pmatrix} = \begin{pmatrix} 0 & 1/2^{e_i} \\ -1/2^{e_i} & ((R_{i-1}/2^{e_{i-1}}) \operatorname{div}_2 R_i)/2^{e_i} \end{pmatrix} \begin{pmatrix} R_{i-1}/2^{e_{i-1}} \\ R_i \end{pmatrix}$$

for each $i \geq 1$ where R_{i+1} is defined.

Proof. We have $(R_{i-1}/2^{e_{i-1}}) \bmod_2 R_i = R_{i-1}/2^{e_{i-1}} - ((R_{i-1}/2^{e_{i-1}}) \operatorname{div}_2 R_i)R_i$. Negate and divide by 2^{e_i} to obtain the matrix equation. □

Theorem E.3. *Let R_0 be an odd element of $\mathbf{Z}$. Let R_1 be an element of $2\mathbf{Z}$. Define $e_0, e_1, e_2, e_3, \ldots$ and $R_2, R_3, \ldots$ as in Theorem E.2. Then $R_i \in 2\mathbf{Z}$ for each $i \geq 1$ where R_i is defined. Furthermore, if $t \geq 0$ and $R_{t+1} = 0$ then $|R_t/2^{e_t}| = \gcd\{R_0, R_1\}$.*

Proof. Write $g = \gcd\{R_0, R_1\} \in \mathbf{Z}$. Then g is odd since R_0 is odd.

We first show by induction on i that $R_i \in g\mathbf{Z}$ for each $i \geq 0$. For $i \in \{0, 1\}$ this follows from the definition of g. For $i \geq 2$, we have $R_{i-2} \in g\mathbf{Z}$ and $R_{i-1} \in g\mathbf{Z}$ by the inductive hypothesis. Now $(R_{i-2}/2^{e_{i-2}}) \operatorname{div}_2 R_{i-1}$ has the form $q/2^{e_{i-1}}$ for $q \in \mathbf{Z}$ by definition of div_2, and $2^{2e_{i-1}+e_{i-2}}R_i = 2^{e_{i-2}}qR_{i-1} - 2^{e_{i-1}}R_{i-2}$ by definition of R_i. The right side of this equation is in $g\mathbf{Z}$, so $2^{\cdots}R_i$ is in $g\mathbf{Z}$. This implies, first, that $R_i \in \mathbf{Q}$, but also $R_i \in \mathbf{Z}_2$, so $R_i \in \mathbf{Z}$. This also implies that $R_i \in g\mathbf{Z}$, since g is odd.

By construction $R_i \in 2\mathbf{Z}_2$ for each $i \geq 1$, and $R_i \in \mathbf{Z}$ for each $i \geq 0$, so $R_i \in 2\mathbf{Z}$ for each $i \geq 1$.

Finally, assume that $t \geq 0$ and $R_{t+1} = 0$. Then all of $R_0, R_1, \ldots, R_t$ are nonzero. Write $g' = |R_t/2^{e_t}|$. Then $2^{e_t}g' = |R_t| \in g\mathbf{Z}$, and g is odd, so $g' \in g\mathbf{Z}$.

The equation $2^{e_{i-1}}R_{i-2} = 2^{e_{i-2}}qR_{i-1} - 2^{2e_{i-1}+e_{i-2}}R_i$ shows that if $R_{i-1}, R_i \in g'\mathbf{Z}$ then $R_{i-2} \in g'\mathbf{Z}$, since g' is odd. By construction $R_t, R_{t+1} \in g'\mathbf{Z}$, so $R_{t-1}, R_{t-2}, \ldots, R_1, R_0 \in g'\mathbf{Z}$. Hence $g = \gcd\{R_0, R_1\} \in g'\mathbf{Z}$. Both g and g' are positive, so $g' = g$. □

E.4. The gcd algorithm. Here is the algorithm featured in this appendix.

Given an odd integer R_0 and an even integer R_1, compute the even integers $R_2, R_3, \ldots$ defined above. In other words, for each $i \geq 1$ where $R_i \neq 0$, compute $e_i = \operatorname{ord}_2 R_i$; find $q \in \{1, 3, 5, \ldots, 2^{e_i+1} - 1\}$ such that $qR_i/2^{e_i} - R_{i-1}/2^{e_{i-1}} \in 2^{e_i+1}\mathbf{Z}$; and compute $R_{i+1} = (qR_i/2^{e_i} - R_{i-1}/2^{e_{i-1}})/2^{e_i} \in 2\mathbf{Z}$. If $R_{t+1} = 0$ for some $t \geq 0$, output $|R_t/2^{e_t}|$ and stop.

We will show in Theorem F.26 that this algorithm terminates. The output is $\gcd\{R_0, R_1\}$ by Theorem E.3. This algorithm is closely related to iterating divstep, and we will use this relationship to analyze iterates of divstep; see Appendix G.

More generally, if R_0 is an odd integer and R_1 is any integer, one can compute $\gcd\{R_0, R_1\}$ by using the above algorithm to compute $\gcd\{R_0, 2R_1\}$. Even more generally,

one can compute the gcd of any two integers by first handling the case $\gcd\{0,0\} = 0$, then handling powers of 2 in both inputs, then applying the odd-input algorithm.

E.5. A non-functioning variant. Imagine replacing the subtractions above with additions: find $q \in \{1, 3, 5, \ldots, 2^{e_i+1} - 1\}$ such that $qR_i/2^{e_i} + R_{i-1}/2^{e_{i-1}} \in 2^{e_i+1}\mathbf{Z}$, and compute $R_{i+1} = (qR_i/2^{e_i} + R_{i-1}/2^{e_{i-1}})/2^{e_i} \in 2\mathbf{Z}$. If R_0 and R_1 are positive then all R_i are positive so the algorithm does not terminate. This non-terminating algorithm is related to posdivstep (see Section 8.4) in the same way that the algorithm above is related to divstep.

Daireaux, Maume-Deschamps, and Vallée in [31, Section 6] mentioned this algorithm as a "non-centered LSB algorithm", said that one can easily add a "supplementary stopping condition" to ensure termination, and dismissed the resulting algorithm as being "certainly slower than the centered version".

E.6. A centered variant: the Stehlé–Zimmermann algorithm. Imagine using centered remainders $\{1 - 2^e, \ldots, -3, -1, 1, 3, \ldots, 2^e - 1\}$ modulo 2^{e+1} rather than uncentered remainders $\{1, 3, 5, \ldots, 2^{e+1} - 1\}$. The above gcd algorithm then turns into the Stehlé–Zimmermann gcd algorithm from [62].

Specifically, Stehlé and Zimmermann begin with integers a, b having $\operatorname{ord}_2 a < \operatorname{ord}_2 b$. If $b \neq 0$ then "generalized binary division" in [62, Lemma 1] defines q as the unique odd integer with $|q| < 2^{\operatorname{ord}_2 b - \operatorname{ord}_2 a}$ such that $r = a + qb/2^{\operatorname{ord}_2 b - \operatorname{ord}_2 a}$ is divisible by $2^{\operatorname{ord}_2 b + 1}$. The gcd algorithm in [62, Algorithm Elementary-GB] repeatedly sets $(a, b) \leftarrow (b, r)$. When $b = 0$, the algorithm outputs the odd part of a.

It is equivalent to set $(a, b) \leftarrow (b/2^{\operatorname{ord}_2 b}, r/2^{\operatorname{ord}_2 b})$, since this simply divides all subsequent results by $2^{\operatorname{ord}_2 b}$. In other words, starting from an odd a_0 and an even b_0, recursively define an odd a_{i+1} and an even b_{i+1} by the following formulas, stopping when $b_i = 0$:

- $a_{i+1} = b_i/2^e$ where $e = \operatorname{ord}_2 b_i$;
- $b_{i+1} = (a_i + qb_i/2^e)/2^e \in 2\mathbf{Z}$ for a unique $q \in \{1 - 2^e, \ldots, -3, -1, 1, 3, \ldots, 2^e - 1\}$.

Now relabel $a_0, b_0, b_1, b_2, b_3, \ldots$ as $R_0, R_1, R_2, R_3, R_4, \ldots$ to obtain the following recursive definition: $R_{i+1} = (qR_i/2^{e_i} + R_{i-1}/2^{e_{i-1}})/2^{e_i} \in 2\mathbf{Z}$, and thus

$$\begin{pmatrix} R_i/2^{e_i} \\ R_{i+1} \end{pmatrix} = \begin{pmatrix} 0 & 1/2^{e_i} \\ 1/2^{e_i} & q/2^{2e_i} \end{pmatrix} \begin{pmatrix} R_{i-1}/2^{e_{i-1}} \\ R_i \end{pmatrix},$$

for a unique $q \in \{1 - 2^{e_i}, \ldots, -3, -1, 1, 3, \ldots, 2^{e_i} - 1\}$.

To summarize, starting from the Stehlé–Zimmermann algorithm, one can obtain the gcd algorithm featured in this appendix by making the following three changes. First, divide out 2^{e_i}, instead of allowing powers of 2 to accumulate. Second, add $R_{i-1}/2^{e_{i-1}}$ instead of subtracting it; this does not affect the number of iterations for centered q. Third, switch from centered q to uncentered q.

Intuitively, centered remainders are smaller than uncentered remainders, and it is well known that centered remainders improve the worst-case performance of Euclid's algorithm, so one might expect that the Stehlé–Zimmermann algorithm performs better than our uncentered variant. However, as noted earlier, our analysis produces the opposite conclusion. See Appendix F.

F Performance of the gcd algorithm for integers

The possible matrices in Theorem E.2 are

$$\begin{pmatrix} 0 & 1/2 \\ -1/2 & 1/4 \end{pmatrix}, \begin{pmatrix} 0 & 1/2 \\ -1/2 & 3/4 \end{pmatrix} \quad \text{for } e_i = 1;$$

$$\begin{pmatrix} 0 & 1/4 \\ -1/4 & 1/16 \end{pmatrix}, \begin{pmatrix} 0 & 1/4 \\ -1/4 & 3/16 \end{pmatrix}, \begin{pmatrix} 0 & 1/4 \\ -1/4 & 5/16 \end{pmatrix}, \begin{pmatrix} 0 & 1/4 \\ -1/4 & 7/16 \end{pmatrix} \quad \text{for } e_i = 2;$$

etc. In this appendix we show that a product of these matrices shrinks the input by a factor exponential in the weight $\sum e_i$. This implies that $\sum e_i$ in the algorithm of Appendix E is $O(b)$, where b is the number of input bits.

F.1. Lower bounds. It is easy to see that each of these matrices has two eigenvalues of absolute value $1/2^{e_i}$. This does not imply, however, that a weight-w product of these matrices shrinks the input by a factor $1/2^w$. Concretely, the weight-7 product

$$\frac{1}{4^7}\begin{pmatrix}328 & -380\\ -158 & 233\end{pmatrix} = \begin{pmatrix}0 & 1/4\\ -1/4 & 1/16\end{pmatrix}\begin{pmatrix}0 & 1/2\\ -1/2 & 3/4\end{pmatrix}^2\begin{pmatrix}0 & 1/2\\ -1/2 & 1/4\end{pmatrix}\begin{pmatrix}0 & 1/2\\ -1/2 & 3/4\end{pmatrix}^2$$

of 6 matrices has spectral radius (maximum absolute value of eigenvalues)

$$(561+\sqrt{249185})/2^{15} = 0.03235425826\ldots = 0.61253803919\ldots^7 = 1/2^{4.949900586\ldots}.$$

The $(w/7)$th power of this matrix is expressed as a weight-w product and has spectral radius $1/2^{0.7071286552\ldots w}$. Consequently, this proof strategy cannot obtain an O constant better than $7/(15-\log_2(561+\sqrt{249185})) = 1.414169815\ldots$. We prove a bound twice the weight on the number of divstep iterations (see Appendix G), so this proof strategy cannot obtain an O constant better than $14/(15-\log_2(561+\sqrt{249185})) = 2.828339631\ldots$ for the number of divstep iterations.

Rotating the list of 6 matrices shown above gives 6 weight-7 products with the same spectral radius. In a small search we did not find any weight-w matrices with spectral radius above $0.61253803919\ldots^w$. Our upper bounds (see below) are 0.6181640625^w for all large w.

For comparison, the non-terminating variant in Appendix E.5 allows the matrix $\begin{pmatrix}0 & 1/2\\ 1/2 & 3/4\end{pmatrix}$, which has spectral radius 1 with eigenvector $\begin{pmatrix}1\\ 2\end{pmatrix}$. The Stehlé–Zimmermann algorithm reviewed in Appendix E.6 allows the matrix $\begin{pmatrix}0 & 1/2\\ 1/2 & 1/4\end{pmatrix}$, which has spectral radius $(1+\sqrt{17})/8 = 0.6403882032\ldots$, so this proof strategy cannot obtain an O constant better than $2/(\log_2(\sqrt{17}-1)-1) = 3.110510062\ldots$ for the number of cdivstep iterations.

F.2. A warning about worst-case inputs. Define M as the matrix $4^{-7}\begin{pmatrix}328 & -380\\ -158 & 233\end{pmatrix}$ shown above. Then $M^{-3} = \begin{pmatrix}60321097 & 113368060\\ 47137246 & 88663112\end{pmatrix}$. Applying our gcd algorithm to inputs $(60321097, 47137246)$ applies a series of 18 matrices of total weight 21: in the notation of Theorem E.2, the matrix $\begin{pmatrix}0 & 1/2\\ -1/2 & 3/4\end{pmatrix}$ twice, then the matrix $\begin{pmatrix}0 & 1/2\\ -1/2 & 1/4\end{pmatrix}$, then the matrix $\begin{pmatrix}0 & 1/2\\ -1/2 & 3/4\end{pmatrix}$ twice, then the weight-2 matrix $\begin{pmatrix}0 & 1/4\\ -1/4 & 1/16\end{pmatrix}$, all of these matrices again, and all of these matrices a third time.

One might think that these are worst-case inputs to our algorithm, analogous to a standard matrix construction of Fibonacci numbers as worst-case inputs to Euclid's algorithm. We emphasize that these are *not* worst-case inputs: the inputs $(22293, -128330)$ are much smaller and nevertheless require 19 steps of total weight 23. We now explain why the analogy fails.

The matrix M^3 has an eigenvector $(1, -0.53\ldots)$ with eigenvalue $1/2^{21\cdot 0.707\ldots} = 1/2^{14.8\ldots}$, and an eigenvector $(1, 0.78\ldots)$ with eigenvalue $1/2^{21\cdot 1.29\ldots} = 1/2^{27.1\ldots}$, so it has spectral radius around $1/2^{14.8}$. Our proof strategy thus cannot guarantee a gain of more than 14.8 bits from weight 21. What we actually show below is that weight 21 is guaranteed to gain more than 13.97 bits. (The quantity α_{21} in Figure F.14 is $133569/2^{31} = 2^{-13.972\ldots}$.)

The matrix M^{-3} has spectral radius $2^{27.1\cdots}$. It is not surprising that each column of M^{-3} is close to 27 bits in size: the only way to avoid this would be for the other eigenvector to be pointing in almost exactly the same direction as $(1,0)$ or $(0,1)$. For our inputs taken from the first column of M^{-3}, the algorithm gains nearly 27 bits from weight 21, almost double the guaranteed gain. In short, these are good inputs, not bad inputs.

Now replace M with the matrix $F = \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}$, which reduces worst-case inputs to Euclid's algorithm. The eigenvalues of F are $(\sqrt{5}-1)/2 = 1/2^{0.69\cdots}$ and $-(\sqrt{5}+1)/2 = -2^{0.69\cdots}$. The entries of powers of F^{-1} are Fibonacci numbers, again with sizes well predicted by the spectral radius of F^{-1}, namely $2^{0.69\cdots}$. However, the spectral radius of F is larger than 1. The reason that this large eigenvalue does not spoil the termination of Euclid's algorithm is that Euclid's algorithm inspects sizes and chooses quotients to decrease sizes at each step.

Stehlé and Zimmermann in [62, Section 6.2] measure the performance of their algorithm for "worst case" inputs[18] obtained from negative powers of the matrix $\begin{pmatrix} 0 & 1/2 \\ 1/2 & 1/4 \end{pmatrix}$. The statement that these inputs are "worst case" is not justified. On the contrary, if these inputs have b bits then they take $(0.73690\ldots + o(1))b$ steps of the Stehlé–Zimmermann algorithm, and thus $(1.4738\ldots + o(1))b$ cdivsteps, which is considerably *fewer* steps than many other inputs that we have tried[19] for various values of b. The quantity $0.73690\ldots$ here is $\log(2)/\log((\sqrt{17}+1)/2)$, coming from the smaller eigenvalue $-2/(\sqrt{17}+1) = -0.39038\ldots$ of the above matrix.

F.3. Norms. We use the standard definitions of the 2-norm of a real vector and the 2-norm of a real matrix. We summarize the basic theory of 2-norms here to keep this paper self-contained.

Definition F.4. *If $v \in \mathbf{R}^2$ then $|v|_2$ is defined as $\sqrt{v_1^2 + v_2^2}$ where $v = (v_1, v_2)$.*

Definition F.5. *If $P \in M_2(\mathbf{R})$ then $|P|_2$ is defined as $\max\{|Pv|_2 : v \in \mathbf{R}^2, |v|_2 = 1\}$.*

This maximum exists because $\{v \in \mathbf{R}^2 : |v|_2 = 1\}$ is a compact set (namely the unit circle) and $|Pv|_2$ is a continuous function of v. See also Theorem F.11 for an explicit formula for $|P|_2$.

Beware that the literature sometimes uses the notation $|P|_2$ for the entrywise 2-norm $\sqrt{P_{11}^2 + P_{12}^2 + P_{21}^2 + P_{22}^2}$. We do not use entrywise norms of matrices in this paper.

Theorem F.6. *Let v be an element of $\mathbf{Z}^2$. If $v \neq 0$ then $|v|_2 \geq 1$.*

Proof. Write v as (v_1, v_2) with $v_1, v_2 \in \mathbf{Z}$. If $v_1 \neq 0$ then $v_1^2 \geq 1$ so $v_1^2 + v_2^2 \geq 1$. If $v_2 \neq 0$ then $v_2^2 \geq 1$ so $v_1^2 + v_2^2 \geq 1$. Either way $|v|_2 = \sqrt{v_1^2 + v_2^2} \geq 1$. □

Theorem F.7. *Let v be an element of $\mathbf{R}^2$. Let r be an element of $\mathbf{R}$. Then $|rv|_2 = |r||v|_2$.*

Proof. Write v as (v_1, v_2) with $v_1, v_2 \in \mathbf{R}$. Then $rv = (rv_1, rv_2)$ so $|rv|_2 = \sqrt{r^2 v_1^2 + r^2 v_2^2} = \sqrt{r^2}\sqrt{v_1^2 + v_2^2} = |r||v|_2$. □

[18] Specifically, Stehlé and Zimmermann claim that "the worst case of the binary variant" (i.e., of their gcd algorithm) is for inputs "G_n and $2G_{n-1}$, where $G_0 = 0$, $G_1 = 1$, $G_n = -G_{n-1} + 4G_{n-2}$, which gives all binary quotients equal to 1". Daireaux, Maume-Deschamps, and Vallée in [31, Section 2.3] state that Stehlé and Zimmermann "exhibited the precise worst-case number of iterations" and give an equivalent characterization of this case.

[19] For example, "G_n" from [62] is -5467345 for $n = 18$ and 2135149 for $n = 17$. The inputs $(-5467345, 2 \cdot 2135149)$ use 17 iterations of the "GB" divisions from [62], while the smaller inputs $(-1287513, 2 \cdot 622123)$ use 24 "GB" iterations. The inputs $(1, -5467345, 2135149)$ use 34 cdivsteps; the inputs $(1, -2718281, 3141592)$ use 45 cdivsteps; the inputs $(1, -1287513, 622123)$ use 58 cdivsteps.

Theorem F.8. *Let P be an element of $M_2(\mathbf{R})$. Let r be an element of $\mathbf{R}$. Then $|rP|_2 = |r||P|_2$.*

Proof. By definition $|rP|_2 = \max\{|rPv|_2 : v \in \mathbf{R}^2, |v|_2 = 1\}$. By Theorem F.7 this is the same as $\max\{|r||Pv|_2\} = |r| \max\{|Pv|_2\} = |r||P|_2$. □

Theorem F.9. *Let P be an element of $M_2(\mathbf{R})$. Let v be an element of $\mathbf{R}^2$. Then $|Pv|_2 \leq |P|_2|v|_2$.*

Proof. If $v = 0$ then $Pv = 0$ and $|Pv|_2 = 0 = |P|_2|v|_2$.

Otherwise $|v|_2 \neq 0$. Write $w = v/|v|_2$. Then $|w|_2 = 1$, so $|Pw|_2 \leq |P|_2$, so $|Pv|_2 = |Pw|_2|v|_2 \leq |P|_2|v|_2$. □

Theorem F.10. *Let P, Q be elements of $M_2(\mathbf{R})$. Then $|PQ|_2 \leq |P|_2|Q|_2$.*

Proof. If $v \in \mathbf{R}^2$ and $|v|_2 = 1$ then $|PQv|_2 \leq |P|_2|Qv|_2 \leq |P|_2|Q|_2|v|_2$. □

Theorem F.11. *Let P be an element of $M_2(\mathbf{R})$. Assume that $P^*P = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ where P^* is the transpose of P. Then $|P|_2 = \sqrt{\frac{a+d+\sqrt{(a-d)^2+4b^2}}{2}}$.*

Proof. $P^*P \in M_2(\mathbf{R})$ is its own transpose, so, by the spectral theorem, it has two orthonormal eigenvectors with real eigenvalues. In other words, $\mathbf{R}^2$ has a basis e_1, e_2 such that

- $|e_1|_2 = 1$,
- $|e_2|_2 = 1$,
- the dot product $e_1^* e_2$ is 0 (here we view vectors as column matrices),
- $P^*Pe_1 = \lambda_1 e_1$ for some $\lambda_1 \in \mathbf{R}$, and
- $P^*Pe_2 = \lambda_2 e_2$ for some $\lambda_2 \in \mathbf{R}$.

Any $v \in \mathbf{R}^2$ with $|v|_2 = 1$ can be written uniquely as $c_1e_1 + c_2e_2$ for some $c_1, c_2 \in \mathbf{R}$ with $c_1^2 + c_2^2 = 1$: explicitly, $c_1 = v^*e_1$ and $c_2 = v^*e_2$. Then $P^*Pv = \lambda_1 c_1 e_1 + \lambda_2 c_2 e_2$.

By definition $|P|_2^2$ is the maximum of $|Pv|_2^2$ over $v \in \mathbf{R}^2$ with $|v|_2 = 1$. Note that $|Pv|_2^2 = (Pv)^*(Pv) = v^*P^*Pv = \lambda_1 c_1^2 + \lambda_2 c_2^2$ with c_1, c_2 defined as above. For example, $0 \leq |Pe_1|_2^2 = \lambda_1$ and $0 \leq |Pe_2|_2^2 = \lambda_2$. Thus $|Pv|_2^2$ achieves $\max\{\lambda_1, \lambda_2\}$. It cannot be larger than this: if $c_1^2 + c_2^2 = 1$ then $\lambda_1 c_1^2 + \lambda_2 c_2^2 \leq \max\{\lambda_1, \lambda_2\}$. Hence $|P|_2^2 = \max\{\lambda_1, \lambda_2\}$.

To finish the proof, we make the spectral theorem explicit for the matrix $P^*P = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \in M_2(\mathbf{R})$. Note that $(P^*P)^* = P^*P$ so $b = c$. The characteristic polynomial of this matrix is $(x-a)(x-d) - b^2 = x^2 - (a+d)x + ad - b^2$, with roots

$$\frac{a+d \pm \sqrt{(a+d)^2 - 4(ad-b^2)}}{2} = \frac{a+d \pm \sqrt{(a-d)^2 + 4b^2}}{2}.$$

The largest eigenvalue of P^*P is thus $(a+d+\sqrt{(a-d)^2+4b^2})/2$, and $|P|_2$ is the square root of this. □

Theorem F.12. *Let P be an element of $M_2(\mathbf{R})$. Assume that $P^*P = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ where P^* is the transpose of P. Let N be an element of $\mathbf{R}$. Then $N \geq |P|_2$ if and only if $N \geq 0$, $2N^2 \geq a + d$, and $(2N^2 - a - d)^2 \geq (a-d)^2 + 4b^2$.*

```
earlybounds = { 0:1, 1:1, 2:689491/2^20, 3:779411/2^21,
  4:880833/2^22, 5:165219/2^20, 6:97723/2^20, 7:882313/2^24,
  8:306733/2^23, 9:92045/2^22, 10:439213/2^25, 11:281681/2^25,
  12:689007/2^27, 13:824303/2^28, 14:257817/2^27, 15:634229/2^29,
  16:386245/2^29, 17:942951/2^31, 18:583433/2^31, 19:713653/2^32,
  20:432891/2^32, 21:133569/2^31, 22:328293/2^33, 23:800421/2^35,
  24:489233/2^35, 25:604059/2^36, 26:738889/2^37, 27:112215/2^35,
  28:276775/2^37, 29:84973/2^36, 30:829297/2^40, 31:253443/2^39,
  32:625405/2^41, 33:95625/2^39, 34:465055/2^42, 35:286567/2^42,
  36:175951/2^42, 37:858637/2^45, 38:65647/2^42, 39:40469/2^42,
  40:24751/2^42, 41:240917/2^46, 42:593411/2^48, 43:364337/2^48,
  44:889015/2^50, 45:543791/2^50, 46:41899/2^47, 47:205005/2^50,
  48:997791/2^53, 49:307191/2^52, 50:754423/2^54, 51:57527/2^51,
  52:281515/2^54, 53:694073/2^56, 54:212249/2^55, 55:258273/2^56,
  56:636093/2^58, 57:781081/2^59, 58:952959/2^60, 59:291475/2^59,
  60:718599/2^61, 61:878997/2^62, 62:534821/2^62, 63:329285/2^62,
  64:404341/2^63, 65:986633/2^65, 66:603553/2^65,
}

def alpha(w):
  if w >= 67: return (633/1024)^w
  return earlybounds[w]

assert all(alpha(w)^49<2^(-(34*w-23)) for w in range(31,100))
assert min((633/1024)^w/alpha(w) for w in range(68))==633^5/(2^30*165219)
```

Figure F.14: Definition of α_w for $w \in \{0,1,2,\ldots\}$; computer verification that $\alpha_w < 2^{-(34w-23)/49}$ for $31 \le w \le 99$; and computer verification that $\min\{(633/1024)^w/\alpha_w : 0 \le w \le 67\} = 633^5/(2^{30}165219)$. If $w \ge 67$ then $\alpha_w = (633/1024)^w$.

Our upper-bound computations work with matrices with rational entries. We use Theorem F.12 to compare the 2-norms of these matrices to various rational bounds N. All of the computations here use exact arithmetic, avoiding the correctness questions that would have shown up if we had used approximations to the square roots in Theorem F.11. Sage includes exact representations of algebraic numbers such as $|P|_2$, but switching to Theorem F.12 made our upper-bound computations an order of magnitude faster.

Proof. Assume that $N \ge 0$; that $2N^2 \ge a+d$; and that $(2N^2-a-d)^2 \ge (a-d)^2+4b^2$. Then $2N^2-a-d \ge \sqrt{(a-d)^2+4b^2}$ since $2N^2-a-d \ge 0$; so $N^2 \ge (a+d+\sqrt{(a-d)^2+4b^2})/2$; so $N \ge \sqrt{(a+d+\sqrt{(a-d)^2+4b^2})/2}$ since $N \ge 0$; so $N \ge |P|_2$ by Theorem F.11.

Conversely, assume that $N \ge |P|_2$. Then $N \ge 0$. Also $N^2 \ge |P|_2^2$, so $2N^2 \ge 2|P|_2^2 = a+d+\sqrt{(a-d)^2+4b^2}$ by Theorem F.11. In particular $2N^2 \ge a+d$, and $(2N^2-a-d)^2 \ge (a-d)^2+4b^2$. □

F.13. Upper bounds. We now show that every weight-w product of the matrices in Theorem E.2 has norm at most α_w. Here $\alpha_0, \alpha_1, \alpha_2, \ldots$ is an exponentially decreasing sequence of positive real numbers defined in Figure F.14.

Definition F.15. *For* $e \in \{1,2,\ldots\}$ *and* $q \in \{1,3,5,\ldots,2^{e+1}-1\}$, *define* $M_{e,q} = \begin{pmatrix} 0 & 1/2^e \\ -1/2^e & q/2^{2e} \end{pmatrix}$.

Theorem F.16. $|M_{e,q}|_2 < (1+\sqrt{2})/2^e$ *if* $e \in \{1,2,\ldots\}$ *and* $q \in \{1,3,5,\ldots,2^{e+1}-1\}$.

Proof. Define $\begin{pmatrix} a & b \\ c & d \end{pmatrix} = M_{e,q}^* M_{e,q}$. Then $a = 1/2^{2e}$, $b = c = -q/2^{3e}$, and $d = 1/2^{2e} + q^2/2^{4e}$, so $a+d = 2/2^{2e} + q^2/2^{4e} < 6/2^{2e}$ and $(a-d)^2 + 4b^2 = 4q^2/2^{6e} + q^4/2^{8e} \le 32/2^{4e}$. By Theorem F.12, $|M_{e,q}|_2 = \sqrt{(a+d+\sqrt{(a-d)^2+4b^2})/2} \le \sqrt{(6/2^{2e} + \sqrt{32}/2^{2e})/2} = (1+\sqrt{2})/2^e$. □

Definition F.17. *Define* $\beta_w = \inf\{\alpha_{w+j}/\alpha_j : j \ge 0\}$ *for* $w \in \{0,1,2,\ldots\}$.

Theorem F.18. *If* $w \in \{0,1,2,\ldots\}$ *then* $\beta_w = \min\{\alpha_{w+j}/\alpha_j : 0 \le j \le 67\}$. *If* $w \ge 67$ *then* $\beta_w = (633/1024)^w 633^5/(2^{30}165219)$.

The first statement reduces the computation of β_w to a finite computation. The computation can be further optimized but is not a bottleneck for us. Similar comments apply to $\gamma_{w,e}$ in Theorem F.20.

Proof. By definition $\alpha_w = (633/1024)^w$ for all $w \ge 67$. If $j \ge 67$ then also $w+j \ge 67$ so $\alpha_{w+j}/\alpha_j = (633/1024)^{w+j}/(633/1024)^j = (633/1024)^w$. In short, all terms α_{w+j}/α_j for $j \ge 67$ are identical, so the infimum of α_{w+j}/α_j for $j \ge 0$ is the same as the minimum for $0 \le j \le 67$.

If $w \ge 67$ then $\alpha_{w+j}/\alpha_j = (633/1024)^{w+j}/\alpha_j$. The quotient $(633/1024)^j/\alpha_j$ for $0 \le j \le 67$ has minimum value $633^5/(2^{30}165219)$. □

Definition F.19. *Define* $\gamma_{w,e} = \inf\{\beta_{w+j}2^j 70/169 : j \ge e\}$ *for* $w \in \{0,1,2,\ldots\}$ *and* $e \in \{1,2,3,\ldots\}$.

This quantity $\gamma_{w,e}$ is designed to be as large as possible subject to two conditions: first, $\gamma_{w,e} \le \beta_{w+e}2^e 70/169$, used in Theorem F.21; second, $\gamma_{w,e} \le \gamma_{w,e+1}$, used in Theorem F.22.

Theorem F.20. $\gamma_{w,e} = \min\{\beta_{w+j}2^j 70/169 : e \le j \le e+67\}$ *if* $w \in \{0,1,2,\ldots\}$ *and* $e \in \{1,2,3,\ldots\}$. *If* $w+e \ge 67$ *then* $\gamma_{w,e} = 2^e(633/1024)^{w+e}(70/169)633^5/(2^{30}165219)$.

Proof. If $w+j \ge 67$ then $\beta_{w+j} = (633/1024)^{w+j}633^5/(2^{30}165219)$ so $\beta_{w+j}2^j 70/169 = (633/1024)^w(633/512)^j(70/169)633^5/(2^{30}165219)$, which increases monotonically with j.

In particular, if $j \ge e+67$ then $w+j \ge 67$ so $\beta_{w+j}2^j 70/169 \ge \beta_{w+e+67}2^{e+67}70/169$. Hence $\gamma_{w,e} = \inf\{\beta_{w+j}2^j 70/169 : j \ge e\} = \min\{\beta_{w+j}2^j 70/169 : e \le j \le e+67\}$. Also, if $w+e \ge 67$ then $w+j \ge 67$ for all $j \ge e$ so

$$\gamma_{w,e} = \left(\frac{633}{1024}\right)^w \left(\frac{633}{512}\right)^e \frac{70}{169}\frac{633^5}{2^{30}165219} = 2^e \left(\frac{633}{1024}\right)^{w+e} \frac{70}{169}\frac{633^5}{2^{30}165219}$$

as claimed. □

Theorem F.21. *Define* S *as the smallest subset of* $\mathbf{Z} \times M_2(\mathbf{Q})$ *such that*

- $\left(0, \left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)\right) \in S$ *and*
- *if* $(w,P) \in S$, *and* $w = 0$ *or* $|P|_2 > \beta_w$, *and* $e \in \{1,2,3,\ldots\}$, *and* $|P|_2 > \gamma_{w,e}$, *and* $q \in \{1,3,5,\ldots,2^{e+1}-1\}$, *then* $(e+w, M(e,q)P) \in S$.

Assume that $|P|_2 \le \alpha_w$ *for each* $(w,P) \in S$. *Then* $|M(e_j,q_j)\cdots M(e_1,q_1)|_2 \le \alpha_{e_1+\cdots+e_j}$ *for all* $j \in \{0,1,2,\ldots\}$, *all* $e_1,\ldots,e_j \in \{1,2,3,\ldots\}$, *and all* $q_1,\ldots,q_j \in \mathbf{Z}$ *such that* $q_i \in \{1,3,5,\ldots,2^{e_i+1}-1\}$ *for each* i.

The idea here is that, instead of searching through all products P of matrices $M(e,q)$ of total weight w to see whether $|P|_2 \le \alpha_w$, we prune the search in a way that makes the search finite across all w (see Theorem F.22), while still being sure to find a minimal counterexample if one exists. The first layer of pruning skips all descendants QP of P if $w > 0$ and $|P|_2 \le \beta_w$; the point is that any such counterexample QP implies a smaller counterexample Q. The second layer of pruning skips weight-$(w+e)$ children $M(e,q)P$ of P if $|P|_2 \le \gamma_{w,e}$; the point is that any such child is eliminated by the first layer of pruning.

Proof. Induct on j.

If $j = 0$ then $M(e_j,q_j)\cdots M(e_1,q_1) = \left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right)$, with norm $1 = \alpha_0$. Assume from now on that $j \ge 1$.

Case 1: $|M(e_i,q_i)\cdots M(e_1,q_1)|_2 \le \beta_{e_1+\cdots+e_i}$ for some $i \in \{1,2,\ldots,j\}$.

Then $j - i < j$, so $|M(e_j,q_j)\cdots M(e_{i+1},q_{i+1})|_2 \le \alpha_{e_{i+1}+\cdots+e_j}$ by the inductive hypothesis, so $|M(e_j,q_j)\cdots M(e_1,q_1)|_2 \le \beta_{e_1+\cdots+e_i}\alpha_{e_{i+1}+\cdots+e_j}$ by Theorem F.10, but $\beta_{e_1+\cdots+e_i}\alpha_{e_{i+1}+\cdots+e_j} \le \alpha_{e_1+\cdots+e_j}$ by definition of β.

Case 2: $|M(e_i,q_i)\cdots M(e_1,q_1)|_2 > \beta_{e_1+\cdots+e_i}$ for every $i \in \{1,2,\ldots,j\}$.

Suppose that $|M(e_{i-1},q_{i-1})\cdots M(e_1,q_1)|_2 \le \gamma_{e_1+\cdots+e_{i-1},e_i}$ for some $i \in \{1,2,\ldots,j\}$. By Theorem F.16, $|M(e_i,q_i)|_2 < (1+\sqrt{2})/2^{e_i} < (169/70)/2^{e_i}$. By Theorem F.10,

$$|M(e_i,q_i)\cdots M(e_1,q_1)|_2 \le |M(e_i,q_i)|_2\gamma_{e_1+\cdots+e_{i-1},e_i} \le \frac{169\gamma_{e_1+\cdots+e_{i-1},e_i}}{70\cdot 2^{e_i+1}} \le \beta_{e_1+\cdots+e_i},$$

contradiction. Thus $|M(e_{i-1},q_{i-1})\cdots M(e_1,q_1)|_2 > \gamma_{e_1+\cdots+e_{i-1},e_i}$ for all $i \in \{1,2,\ldots,j\}$.

Now $(e_1+\cdots+e_i, M(e_i,q_i)\cdots M(e_1,q_1)) \in S$ for each $i \in \{0,1,2,\ldots,j\}$. Indeed, induct on i. The base case $i = 0$ is simply $\left(0, \left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right)\right) \in S$. If $i \ge 1$ then $(w,P) \in S$ by the inductive hypothesis, where $w = e_1+\cdots+e_{i-1}$ and $P = M(e_{i-1},q_{i-1})\cdots M(e_1,q_1)$. Furthermore

- $w = 0$ (when $i = 1$) or $|P|_2 > \beta_w$ (when $i \ge 2$, by definition of this case);
- $e_i \in \{1,2,3,\ldots\}$;
- $|P|_2 > \gamma_{w,e_i}$ (as shown above); and
- $q_i \in \{1,3,5,\ldots,2^{e_i+1}-1\}$.

Hence $(e_1+\cdots+e_i, M(e_i,q_i)\cdots M(e_1,q_1)) = (e_i+w, M(e_i,q_i)P) \in S$ by definition of S.

In particular, $(w,P) \in S$ with $w = e_1+\cdots+e_j$ and $P = M(e_j,q_j)\cdots M(e_1,q_1)$, so $|P|_2 \le \alpha_w$ by hypothesis. □

Theorem F.22. *In Theorem F.21, S is finite, and $|P|_2 \le \alpha_w$ for each $(w,P) \in S$.*

Proof. This is a computer proof. We run the Sage script in Figure F.23. We observe that it terminates successfully (in slightly over 2546 minutes using Sage 8.6 on one core of a 3.5GHz Intel Xeon E3-1275 v3 CPU) and prints 3787975117.

What the script does is search, depth-first, through the elements of S, verifying that each $(w,P) \in S$ satisfies $|P|_2 \le \alpha_w$. The output is the number of elements searched, so S has at most 3787975117 elements.

Specifically, if $(w,P) \in S$, then `verify(w,P)` checks that $|P|_2 \le \alpha_w$, and recursively calls `verify` on all descendants of (w,P). Actually, for efficiency, the script scales each matrix to have integer entries: it works with $2^{2e}M(e,q)$ (labeled `scaledM(e,q)` in the script) instead of $M(e,q)$, and it works with 4^wP (labeled `P` in the script) instead of P.

The script computes β_w as shown in Theorem F.18, computes γ_w as shown in Theorem F.20, and compares $|P|_2$ to various bounds as shown in Theorem F.12.

If $w > 0$ and $|P|_2 \le \beta_w$ then `verify(w,P)` returns. There are no descendants of (w,P) in this case. Also $|P|_2 \le \alpha_w$, since $\beta_w \le \alpha_w/\alpha_0 = \alpha_w$.

```
from alpha import alpha
from memoized import memoized

R = MatrixSpace(ZZ,2)
def scaledM(e,q):
  return R((0,2^e,-2^e,q))

@memoized
def beta(w):
  return min(alpha(w+j)/alpha(j) for j in range(68))

@memoized
def gamma(w,e):
  return min(beta(w+j)*2^j*70/169 for j in range(e,e+68))

def spectralradiusisatmost(PP,N): # assuming PP has form P.transpose()*P
  (a,b),(c,d) = PP
  X = 2*N^2-a-d
  return N >= 0 and X >= 0 and X^2 >= (a-d)^2+4*b^2

def verify(w,P):
  nodes = 1
  PP = P.transpose()*P
  if w>0 and spectralradiusisatmost(PP,4^w*beta(w)): return nodes
  assert spectralradiusisatmost(PP,4^w*alpha(w))
  for e in PositiveIntegers():
    if spectralradiusisatmost(PP,4^w*gamma(w,e)): return nodes
    for q in range(1,2^(e+1),2):
      nodes += verify(e+w,scaledM(e,q)*P)

print verify(0,R(1))
```

Figure F.23: Sage script verifying $|P|_2 \leq \alpha_w$ for each $(w, P) \in S$. Output is number of pairs (w, P) considered if verification succeeds, or an assertion failure if verification fails.

Otherwise `verify(w,P)` asserts that $|P|_2 \leq \alpha_w$: i.e., it checks that $|P|_2 \leq \alpha_w$, and raises an exception if not. It then tries successively $e = 1$, $e = 2$, $e = 3$, etc., continuing as long as $|P|_2 > \gamma_{w,e}$. For each e where $|P|_2 > \gamma_{w,e}$, `verify(w,P)` recursively handles $(e + w, M(e, q)P)$ for each $q \in \{1, 3, 5, \ldots, 2^{e+1} - 1\}$.

Note that $\gamma_{w,e} \leq \gamma_{w,e+1} \leq \cdots$, so if $|P|_2 \leq \gamma_{w,e}$ then also $|P|_2 \leq \gamma_{w,e+1}$ etc. This is where it is important that $\gamma_{w,e}$ is not simply defined as $\beta_{w+e}2^e 70/169$.

To summarize, `verify(w,P)` recursively enumerates all descendants of (w, P). Hence `verify(0,R(1))` recursively enumerates all elements (w, P) of S, checking $|P|_2 \leq \alpha_w$ for each (w, P). □

Theorem F.24. *If $j \in \{0, 1, 2, \ldots\}$, $e_1, \ldots, e_j \in \{1, 2, 3, \ldots\}$, $q_1, \ldots, q_j \in \mathbf{Z}$, and $q_i \in \{1, 3, 5, \ldots, 2^{e_i+1} - 1\}$ for each i, then $|M(e_j, q_j) \cdots M(e_1, q_1)|_2 \leq \alpha_{e_1+\cdots+e_j}$.*

Proof. This is the conclusion of Theorem F.21, so it suffices to check the assumptions. Define S as in Theorem F.21; then $|P|_2 \leq \alpha_w$ for each $(w, P) \in S$ by Theorem F.22. □

Theorem F.25. *In the situation of Theorem E.3, assume that $R_0, R_1, R_2, \ldots, R_{j+1}$ are defined. Then $|(R_j/2^{e_j}, R_{j+1})|_2 \le \alpha_{e_1+\cdots+e_j}|(R_0, R_1)|_2$, and if $e_1 + \cdots + e_j \ge 67$ then $e_1 + \cdots + e_j \le \log_{1024/633} |(R_0, R_1)|_2$.*

If R_0 and R_1 are each bounded by 2^b in absolute value then $|(R_0, R_1)|_2$ is bounded by $2^{b+0.5}$ so $\log_{1024/633} |(R_0, R_1)|_2 \le (b + 0.5)/\log_2(1024/633) = (b + 0.5)(1.4410\ldots)$.

Proof. By Theorem E.2,

$$\begin{pmatrix} R_i/2^{e_i} \\ R_{i+1} \end{pmatrix} = M(e_i, q_i) \begin{pmatrix} R_{i-1}/2^{e_{i-1}} \\ R_i \end{pmatrix}$$

where $q_i = 2_i^e((R_{i-1}/2^{e_{i-1}}) \operatorname{div}_2 R_i) \in \{1, 3, 5, \ldots, 2^{e_i+1} - 1\}$. Hence

$$\begin{pmatrix} R_j/2^{e_j} \\ R_{j+1} \end{pmatrix} = M(e_j, q_j) \cdots M(e_1, q_1) \begin{pmatrix} R_0 \\ R_1 \end{pmatrix}.$$

The product $M(e_j, q_j) \cdots M(e_1, q_1)$ has 2-norm at most α_w where $w = e_1 + \cdots + e_j$, so $|(R_j/2^{e_j}, R_{j+1})|_2 \le \alpha_w |(R_0, R_1)|_2$ by Theorem F.9.

By Theorem F.6, $|(R_j/2^{e_j}, R_{j+1})|_2 \ge 1$. If $e_1 + \cdots + e_j \ge 67$ then $\alpha_{e_1+\cdots+e_j} = (633/1024)^{e_1+\cdots+e_j}$ so $1 \le (633/1024)^{e_1+\cdots+e_j} |(R_0, R_1)|_2$. □

Theorem F.26. *In the situation of Theorem E.3, there exists $t \ge 0$ such that $R_{t+1} = 0$.*

Proof. Suppose not. Then $R_0, R_1, \ldots, R_j, R_{j+1}$ are all defined for arbitrarily large j. Take $j \ge 67$ with $j > \log_{1024/633} |(R_0, R_1)|_2$. Then $e_1 + \cdots + e_j \ge 67$ so $j \le e_1 + \cdots + e_j \le \log_{1024/633} |(R_0, R_1)|_2 < j$ by Theorem F.25, contradiction. □

G Proof of main gcd theorem for integers

This appendix shows how the gcd algorithm from Appendix E is related to iterating divstep. This appendix then uses the analysis from Appendix F to put bounds on the number of divstep iterations needed.

Theorem G.1 (2-adic division as a sequence of division steps)**.** *In the situation of Theorem E.1,* $\operatorname{divstep}^{2e}(1, f, g/2) = (1, g/2^e, (qg/2^e - f)/2^{e+1})$.

In other words, $\operatorname{divstep}^{2e}(1, f, g/2) = (1, g/2^e, -(f \bmod_2 g)/2^{e+1})$.

Proof. Define

$$\begin{aligned} z_e &= (g/2^e - f)/2 \in \mathbf{Z}_2, \\ z_{e+1} &= (z_e + (z_e \bmod 2)g/2^e)/2 \in \mathbf{Z}_2, \\ z_{e+2} &= (z_{e+1} + (z_{e+1} \bmod 2)g/2^e)/2 \in \mathbf{Z}_2, \\ &\vdots \\ z_{2e} &= (z_{2e-1} + (z_{2e-1} \bmod 2)g/2^e)/2 \in \mathbf{Z}_2. \end{aligned}$$

Then

$$\begin{aligned} z_e &= (g/2^e - f)/2, \\ z_{e+1} &= ((1 + 2(z_e \bmod 2))g/2^e - f)/4, \\ z_{e+2} &= ((1 + 2(z_e \bmod 2) + 4(z_{e+1} \bmod 2))g/2^e - f)/8, \\ &\vdots \\ z_{2e} &= ((1 + 2(z_e \bmod 2) + \cdots + 2^e(z_{2e-1} \bmod 2))g/2^e - f)/2^{e+1}. \end{aligned}$$

In short, $z_{2e} = (qg/2^e - f)/2^{e+1}$ where

$$q' = 1 + 2(z_e \bmod 2) + \cdots + 2^e(z_{2e-1} \bmod 2) \in \{1, 3, 5, \ldots, 2^{e+1} - 1\}.$$

Hence $q'g/2^e - f = 2^{e+1}z_{2e} \in 2^{e+1}\mathbf{Z}_2$, so $q' = q$ by the uniqueness statement in Theorem E.1. Note that this construction gives an alternative proof of the existence of q in Theorem E.1.

All that remains is to prove $\operatorname{divstep}^{2e}(1, f, g/2) = (1, g/2^e, (qg/2^e - f)/2^{e+1})$, i.e., $\operatorname{divstep}^{2e}(1, f, g/2) = (1, g/2^e, z_{2e})$.

If $1 \le i < e$ then $g/2^i \in 2\mathbf{Z}_2$ so $\operatorname{divstep}(i, f, g/2^i) = (i+1, f, g/2^{i+1})$. By induction, $\operatorname{divstep}^{e-1}(1, f, g/2) = (e, f, g/2^e)$. By assumption $e > 0$ and $g/2^e$ is odd so $\operatorname{divstep}(e, f, g/2^e) = (1-e, g/2^e, (g/2^e - f)/2) = (1-e, g/2^e, z_e)$. What remains is to prove that $\operatorname{divstep}^e(1-e, g/2^e, z_e) = (1, g/2^e, z_{2e})$.

If $0 \le i \le e-1$ then $1-e+i \le 0$ so $\operatorname{divstep}(1-e+i, g/2^e, z_{e+i}) = (2-e+i, g/2^e, (z_{e+i} + (z_{e+i} \bmod 2)g/2^e)/2) = (2-e+i, g/2^e, z_{e+i+1})$. By induction, $\operatorname{divstep}^i(1-e, g/2^e, z_e) = (1-e+i, g/2^e, z_{e+i})$ for $0 \le i \le e$. In particular, $\operatorname{divstep}^e(1-e, g/2^e, z_e) = (1, g/2^e, z_{2e})$ as claimed. □

Theorem G.2 (2-adic gcd as a sequence of division steps)**.** *In the situation of Theorem E.3, assume that $R_0, R_1, \ldots, R_j, R_{j+1}$ are defined. Then* $\operatorname{divstep}^{2w}(1, R_0, R_1/2) = (1, R_j/2^{e_j}, R_{j+1}/2)$ *where* $w = e_1 + \cdots + e_j$.

Therefore, if $R_{t+1} = 0$, then $\operatorname{divstep}^n(1, R_0, R_1/2) = (1 + n - 2w, R_t/2^{e_t}, 0)$ for all $n \ge 2w$, where $w = e_1 + \cdots + e_t$.

Proof. Induct on j. If $j = 0$ then $w = 0$ so $\operatorname{divstep}^{2w}(1, R_0, R_1/2) = (1, R_0, R_1/2)$, and $R_0 = R_0/2^{e_0}$ since R_0 is odd by hypothesis.

If $j \ge 1$ then by definition $R_{j+1} = -((R_{j-1}/2^{e_{j-1}}) \bmod_2 R_j)/2^{e_j}$. Furthermore

$$\operatorname{divstep}^{2e_1+\cdots+2e_{j-1}}(1, R_0, R_1/2) = (1, R_{j-1}/2^{e_{j-1}}, R_j/2)$$

by the inductive hypothesis, so

$$\operatorname{divstep}^{2e_1+\cdots+2e_j}(1, R_0, R_1/2) = \operatorname{divstep}^{2e_j}(1, R_{j-1}/2^{e_{j-1}}, R_j/2) = (1, R_j/2^{e_j}, R_{j+1}/2)$$

by Theorem G.1. □

Theorem G.3. *Let R_0 be an odd element of $\mathbf{Z}$. Let R_1 be an element of $2\mathbf{Z}$. Then there exists $n \in \{0, 1, 2, \ldots\}$ such that* $\operatorname{divstep}^n(1, R_0, R_1/2) \in \mathbf{Z} \times \mathbf{Z} \times \{0\}$. *Furthermore, any such n has* $\operatorname{divstep}^n(1, R_0, R_1/2) \in \mathbf{Z} \times \{G, -G\} \times \{0\}$ *where* $G = \gcd\{R_0, R_1\}$.

Proof. Define $e_0, e_1, e_2, e_3, \ldots$ and $R_2, R_3, \ldots$ as in Theorem E.2. By Theorem F.26, there exists $t \ge 0$ such that $R_{t+1} = 0$. By Theorem E.3, $R_t/2^{e_t} \in \{G, -G\}$. By Theorem G.2, $\operatorname{divstep}^{2w}(1, R_0, R_1/2) = (1, R_t/2^{e_t}, 0) \in \mathbf{Z} \times \{G, -G\} \times \{0\}$ where $w = e_1 + \cdots + e_t$.

Conversely, assume that $n \in \{0, 1, 2, \ldots\}$ and that $\operatorname{divstep}^n(1, R_0, R_1/2) \in \mathbf{Z} \times \mathbf{Z} \times \{0\}$. Write $\operatorname{divstep}^n(1, R_0, R_1/2)$ as $(\delta, f, 0)$. Then $\operatorname{divstep}^{n+k}(1, R_0, R_1/2) = (k + \delta, f, 0)$ for all $k \in \{0, 1, 2, \ldots\}$, so in particular $\operatorname{divstep}^{2w+n}(1, R_0, R_1/2) = (2w + \delta, f, 0)$. Also $\operatorname{divstep}^{2w+k}(1, R_0, R_1/2) = (k+1, R_t/2^{e_t}, 0)$ for all $k \in \{0, 1, 2, \ldots\}$, so in particular $\operatorname{divstep}^{2w+n}(1, R_0, R_1/2) = (n+1, R_t/2^{e_t}, 0)$. Hence $f = R_t/2^{e_t} \in \{G, -G\}$, and $\operatorname{divstep}^n(1, R_0, R_1/2) \in \mathbf{Z} \times \{G, -G\} \times \{0\}$ as claimed. □

Theorem G.4. *Let R_0 be an odd element of $\mathbf{Z}$. Let R_1 be an element of $2\mathbf{Z}$. Define $b = \log_2 \sqrt{R_0^2 + R_1^2}$. Assume that $b \le 21$. Then* $\operatorname{divstep}^{\lfloor 19b/7 \rfloor}(1, R_0, R_1/2) \in \mathbf{Z} \times \mathbf{Z} \times \{0\}$.

Proof. If $\operatorname{divstep}(\delta, f, g) = (\delta_1, f_1, g_1)$ then $\operatorname{divstep}(\delta, -f, -g) = (\delta_1, -f_1, -g_1)$. Hence $\operatorname{divstep}^n(1, R_0, R_1/2) \in \mathbf{Z} \times \mathbf{Z} \times \{0\}$ if and only if $\operatorname{divstep}^n(1, -R_0, -R_1/2) \in \mathbf{Z} \times \mathbf{Z} \times \{0\}$. From now on assume without loss of generality that $R_0 > 0$.

s	W_s	R_0	R_1
0	1	1	0
1	5	1	−2
2	5	1	−2
3	5	1	−2
4	17	1	−4
5	17	1	−4
6	65	1	−8
7	65	1	−8
8	157	11	−6
9	181	9	−10
10	421	15	−14
11	541	21	−10
12	1165	29	−18
13	1517	29	−26
14	2789	17	−50
15	3653	47	−38
16	8293	47	−78
17	8293	47	−78
18	24245	121	−98
19	27361	55	−156
20	56645	191	−142
21	79349	215	−182
22	132989	283	−230
23	213053	133	−442
24	415001	451	−460
25	613405	501	−602
26	1345061	719	−910
27	1552237	1021	−714
28	2866525	789	−1498

s	W_s	R_0	R_1
29	4598269	1355	−1662
30	7839829	777	−2690
31	12565573	2433	−2578
32	22372085	2969	−3682
33	35806445	5443	−2486
34	71013905	4097	−7364
35	74173637	5119	−6926
36	205509533	9973	−10298
37	226964725	11559	−9662
38	487475029	11127	−19070
39	534274997	13241	−18946
40	1543129037	24749	−30506
41	1639475149	20307	−35030
42	3473731181	39805	−43466
43	4500780005	49519	−45262
44	9497198125	38051	−89718
45	12700184149	82743	−76510
46	29042662405	152287	−76494
47	36511782869	152713	−114850
48	82049276437	222249	−180706
49	90638999365	220417	−205074
50	234231548149	229593	−426050
51	257130797053	338587	−377478
52	466845672077	301069	−613354
53	622968125533	437163	−657158
54	1386285660565	600809	−1012578
55	1876581808109	604547	−1229270
56	4208169535453	1352357	−1542498

Figure G.5: For each $s \in \{0, 1, \ldots, 56\}$: Minimum $R_0^2 + R_1^2$ where R_0 is an odd integer, R_1 is an even integer, and (R_0, R_1) requires $\geq s$ iterations of divstep. Also, an example of (R_0, R_1) reaching this minimum.

The rest of the proof is a computer proof. We enumerate all pairs (R_0, R_1) where R_0 is a positive odd integer, R_1 is an even integer, and $R_0^2 + R_1^2 \leq 2^{42}$. For each (R_0, R_1), we iterate divstep to find the minimum $n \geq 0$ where $\text{divstep}^n(1, R_0, R_1/2) \in \mathbf{Z} \times \mathbf{Z} \times \{0\}$. We then say that (R_0, R_1) "needs n steps".

For each $s \geq 0$, define W_s as the smallest $R_0^2 + R_1^2$ where (R_0, R_1) needs $\geq s$ steps. The computation shows that each (R_0, R_1) needs ≤ 56 steps, and also shows the values W_s for each $s \in \{0, 1, \ldots, 56\}$. We display these values in Figure G.5. We check, for each $s \in \{0, 1, \ldots, 56\}$, that $2^{14s} \leq W_s^{19}$.

Finally, we claim that each (R_0, R_1) needs $\leq \lfloor 19b/7 \rfloor$ steps where $b = \log_2 \sqrt{R_0^2 + R_1^2}$. If not then (R_0, R_1) needs $\geq s$ steps where $s = \lfloor 19b/7 \rfloor + 1$. We have $s \leq 56$, since (R_0, R_1) needs ≤ 56 steps. We also have $W_s \leq R_0^2 + R_1^2$, by definition of W_s. Hence $2^{14s/19} \leq R_0^2 + R_1^2$, so $14s/19 \leq 2b$, so $s \leq 19b/7$, contradiction. □

Theorem G.6 (Bounds on the number of divsteps required)**.** *Let R_0 be an odd element of $\mathbf{Z}$. Let R_1 be an element of $2\mathbf{Z}$. Define $b = \log_2 \sqrt{R_0^2 + R_1^2}$. Define $n = \lfloor 19b/7 \rfloor$ if $b \leq 21$; $n = \lfloor (49b + 23)/17 \rfloor$ if $21 < b \leq 46$; and $n = \lfloor 49b/17 \rfloor$ if $b > 46$. Then* $\text{divstep}^n(1, R_0, R_1/2) \in \mathbf{Z} \times \mathbf{Z} \times \{0\}$.

All three cases have $n \le \lfloor (49b+23)/17 \rfloor$ since $19/7 < 49/17$.

Proof. If $b \le 21$ then $\text{divstep}^n(1, R_0, R_1/2) \in \mathbf{Z} \times \mathbf{Z} \times \{0\}$ by Theorem G.4.

Assume from now on that $b > 21$. This implies $n \ge \lfloor 49b/17 \rfloor \ge 60$.

Define $e_0, e_1, e_2, e_3, \ldots$ and $R_2, R_3, \ldots$ as in Theorem E.2. By Theorem F.26, there exists $t \ge 0$ such that $R_{t+1} = 0$. By Theorem E.3, $R_t/2^{e_t} \in \{G, -G\}$. By Theorem G.2, $\text{divstep}^{2w}(1, R_0, R_1/2) = (1, R_t/2^{e_t}, 0) \in \mathbf{Z} \times \{G, -G\} \times \{0\}$ where $w = e_1 + \cdots + e_t$.

To finish the proof we will show that $2w \le n$. Hence $\text{divstep}^n(1, R_0, R_1/2) \in \mathbf{Z} \times \mathbf{Z} \times \{0\}$ as claimed.

Suppose that $2w > n$. Both $2w$ and n are integers, so $2w \ge n+1 \ge 61$, so $w \ge 31$.

By Theorem F.6 and Theorem F.25, $1 \le |(R_t/2^{e_t}, R_{t+1})|_2 \le \alpha_w |(R_0, R_1)|_2 = \alpha_w 2^b$ so $\log_2(1/\alpha_w) \le b$.

Case 1: $w \ge 67$. By definition $1/\alpha_w = (1024/633)^w$ so $w \log_2(1024/633) \le b$. We have $34/49 < \log_2(1024/633)$ so $34w/49 < b$ so $n+1 \le 2w < 49b/17 < (49b+23)/17$. This contradicts the definition of n as the largest integer below $49b/17$ if $b > 46$, and as the largest integer below $(49b+23)/17$ if $b \le 46$.

Case 2: $w \le 66$. Then $\alpha_w < 2^{-(34w-23)/49}$ since $w \ge 31$, so $b \ge \lg(1/\alpha_w) > (34w-23)/49$, so $n+1 \le 2w \le (49b+23)/17$. This again contradicts the definition of n if $b \le 46$. If $b > 46$ then $n \ge \lfloor 49 \cdot 46/17 \rfloor = 132$ so $2w \ge n+1 > 132 \ge 2w$, contradiction. □

H Comparison to performance of cdivstep

This appendix briefly compares the analysis of divstep from Appendix G to an analogous analysis of cdivstep.

First modify Theorem E.1 to take the unique element $q \in \{\pm 1, \pm 3, \ldots, \pm(2^e - 1)\}$ such that $f + qg/2^e \in 2^{e+1}\mathbf{Z}_2$. The corresponding modification of Theorem G.1 says that $\text{cdivstep}^{2e}(1, f, g/2) = (1, g/2^e, (f + qg/2^e)/2^{e+1})$. The proof proceeds identically except $z_j = (z_{j-1} - (z_{j-1} \bmod 2)g/2^e)/2 \in \mathbf{Z}_2$ for $e < j < 2e$ and $z_{2e} = (z_{2e-1} + (z_{2e-1} \bmod 2)g/2^e)/2 \in \mathbf{Z}_2$. Also we have $q = -1 - 2(z_e \bmod 2) - \cdots - 2^{e-1}(z_{2e-2} \bmod 2) + 2^e(z_{2e-1} \bmod 2) \in \{\pm 1, \pm 3, \ldots, \pm(2^e - 1)\}$. In the notation of [62], $\text{GB}(f, g) = (q, r)$ where $r = f + qg/2^e = 2^{e+1} z_{2e}$.

The corresponding gcd algorithm is the centered algorithm from Appendix E.6, with addition instead of subtraction. Recall that, except for inessential scalings by powers of 2, this is the same as the Stehlé–Zimmermann gcd algorithm from [62].

The corresponding set of transition matrices is different from the divstep case. As mentioned in Appendix F, there are weight-w products with spectral radius $0.6403882032\ldots^w$, so the proof strategy for cdivstep cannot obtain weight-w norm bounds below this spectral radius. This is worse than our bounds for divstep.

Enumerating small pairs (R_0, R_1), as in Theorem G.4, also produces worse results for cdivstep than for divstep. See Figure H.1.

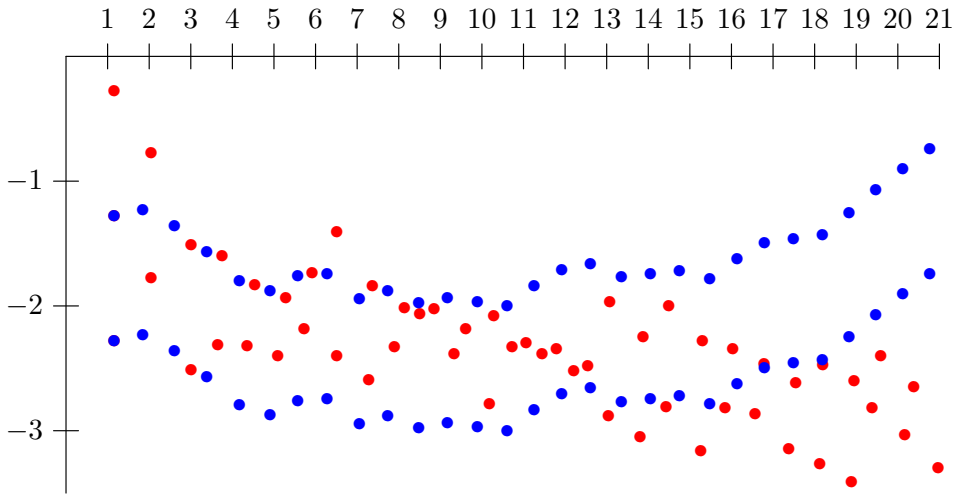

Figure H.1: For each $s \in \{1, 2, \ldots, 56\}$: red dot at $(b, s - 2.8283396b)$ where b is minimum $\log_2 \sqrt{R_0^2 + R_1^2}$ where R_0 is an odd integer, R_1 is an even integer, and (R_0, R_1) requires $\geq s$ iterations of divstep. For each $s \in \{1, 2, \ldots, 58\}$: blue dot at $(b, s - 2.8283396b)$ where b is minimum $\log_2 \sqrt{R_0^2 + R_1^2}$ where R_0 is an odd integer, R_1 is an even integer, and (R_0, R_1) requires $\geq s$ iterations of cdivstep.