FFT extension for algebraic-group factorization algorithms

Richard Brent, Alexander Kruppa, Paul Zimmermann

1

FFT extension for algebraic-group factorization algorithms

Richard P. Brent, Alexander Kruppa and Paul Zimmermann

It is well known that the second stage of factoring methods that exploit smoothness of group orders can be implemented efficiently using the fast Fourier transform (FFT). For Pollard's $p-1$ method [17] this originated with the Montgomery and Silverman paper [16], and for the elliptic curve factoring method [12] it was the subject of Peter Montgomery's PhD dissertation [14]. Along with Peter's most recent work on this subject [15], these developments are presented in this chapter.

1.1 Introduction

We assume that the reader is familiar with the basic ideas of the Elliptic Curve Method of factorization (ECM) – an overview is given in the survey paper [19] and a high-level description is given in Section 1.2 on page 3. ECM was invented by H. W. Lenstra, Jr., and information about it circulated informally as early as February 1985, although Lenstra's *Annals* paper [12] did not appear in print until 1987. Already by the end of 1985 Peter Montgomery and the first author had implemented ECM and found various practical improvements described in [7, 13, 19]. The most significant improvements were:

1 Implementation of a "stage two" analogous to the stage two in Pollard's $p-1$ method [13, 17]. Several variants of stage two were suggested, see [19].

2 Use of Montgomery's form of elliptic curve, given here in homogeneous form where the triple $(x : y : z)$ represents the point $(x/z : y/z)$ in affine coordinates:

$$E_{a,b} : by^2z = x^3 + ax^2z + xz^2$$

instead of the classical (homogeneous) Weierstrass form

$$y^2z = x^3 + axz^2 + bz^3.$$

This uses fewer operations in $\mathbb{Z}/n\mathbb{Z}$, where n is the number to factor, although it introduces the complication of Lucas chains. In stage one it is faster to use Montgomery's form than the Weierstrass form.[1] In stage two it may be better to switch to the homogeneous Weierstrass form. However, stage one typically takes more than 50% of the running time, so the improvement due to using Montgomery's form in stage one is significant.

3 Montgomery's form ensures that the group order g is divisible by 4; an improvement is Suyama's parametrization which ensures that g is divisible by 12. In Chapter 6 of his dissertation [14], Montgomery considers various ways of ensuring that g is divisible by 12 or 16 (subject to a congruence condition).[2]

We remark that these improvements, although significant in practice, do not change the theoretical order of the (conjectured) expected running time

$$O\left(L(p)^{\sqrt{2}+o(1)}M_n\right)$$

to find a prime factor p of n, where $L(p) = \exp(\sqrt{\ln p \ln \ln p})$, and M_n represents the complexity of multiplication modulo n. All the improvements can be absorbed into the $o(1)$ term, so asymptotically they are unimportant. However, in practice they are highly significant.

Other improvements were introduced later. A survey of developments up to about 2006 is given in [19]. The most significant of these later improvements is the "FFT extension" which was proposed by Montgomery around 1990 and published in his dissertation [14]. The idea of using the FFT had already been suggested by Pollard [17] for his $p-1$ method (see also [16]), but Montgomery was the first to see how to apply the FFT to speed up ECM. We note that "FFT extension" is a poor name – more accurate would be "fast polynomial arithmetic extension/continuation" – since any algorithm for fast polynomial multiplication can be used, not just the FFT. However, "FFT extension" has been used consistently in the literature, so to avoid confusion we adopt this terminology.

In Section 1.2 we describe Montgomery's FFT extension for ECM, and in Section 1.3 on page 11 we describe the FFT extension for Pollard's $p-1$

[1] See Bernstein and Lange [2] for a discussion of an alternative form suggested recently by Edwards [9].

[2] See also the erratum to [19].

method and Williams' $p+1$ method [18], where some additional speedups are possible.

1.2 FFT extension for the elliptic curve method

To fix our notation, we first give a high-level description of ECM. Suppose that the stage one and stage two limits are B_1 and B_2, respectively. In the following π denotes a rational prime.

Stage one of ECM takes a point $P_0 = (x_0 : y_0 : z_0)$ on a pseudo-random elliptic curve $E_{a,b}$ mod n (ignoring the fact that n is not a prime!), and computes

$$Q = \prod_{\pi \le B_1} \pi^{e(B_1,\pi)} P_0$$

on $E_{a,b}$, where $e(B_1, \pi) = \lfloor \ln B_1 / \ln \pi \rfloor$. (If the computation breaks down then most likely a GCD computation will give a non-trivial factor of n – see [19] for details.) Assuming that there was no breakdown in the computation of Q, stage two computes, for each (prime) $\pi \in (B_1, B_2]$,

$$(x_\pi : y_\pi : z_\pi) = \pi Q \text{ on } E_{a,b} \text{ and } g_\pi = \gcd(n, z_\pi),$$

and if $g_\pi \neq 1$, outputs g_π (a factor of n) and exits. If all the g_π are 1 then stage two fails and we restart stage one with another choice of pseudo-random elliptic curve.

If implemented as we have just described, stage two takes $\Theta(B_2/\ln B_2)$ operations, assuming $B_1 \ll B_2$. Montgomery's idea is to use fast polynomial arithmetic to reduce this complexity.

We take two sets S and T such that $S + T = \{\sigma + \tau : \sigma \in S, \tau \in T\}$ covers all primes in $(B_1, B_2]$. For example: $d = \lfloor B_2^{1/2} \rfloor$, $S = \{i \cdot d : 0 \le i \cdot d \le B_2\}$, $T = \{j : 0 < j < d, \gcd(j, d) = 1\}$. Computing all values of σQ and τQ costs $O(B_2/d + d)$ operations, which is $O(B_2^{1/2})$ by our choice of d. Also, all primes $\pi \in (d, B_2]$ can be written as $\pi = \sigma + \tau$ for some $\sigma \in S$ and $\tau \in T$.

Assume that we use affine coordinates for stage two, so $\sigma Q = (x_\sigma : y_\sigma)$ and $\tau Q = (x_\tau : y_\tau)$. Then $\sigma Q + \tau Q = O_E$ mod p implies that $x_\sigma = x_\tau$ mod p (where O_E is the neutral element in the group of the elliptic curve mod p, and p is a prime factor of n). Thus, it suffices to compute $\gcd(x_\sigma - x_\tau, n)$ to obtain the factor p (except in the unlikely case that $x_\sigma = x_\tau$ mod n; we shall ignore this possibility).

Thus, the problem reduces to computing

$$h = \prod_{\sigma \in S} \prod_{\tau \in T} (x_\sigma - x_\tau) \bmod n, \tag{1.1}$$

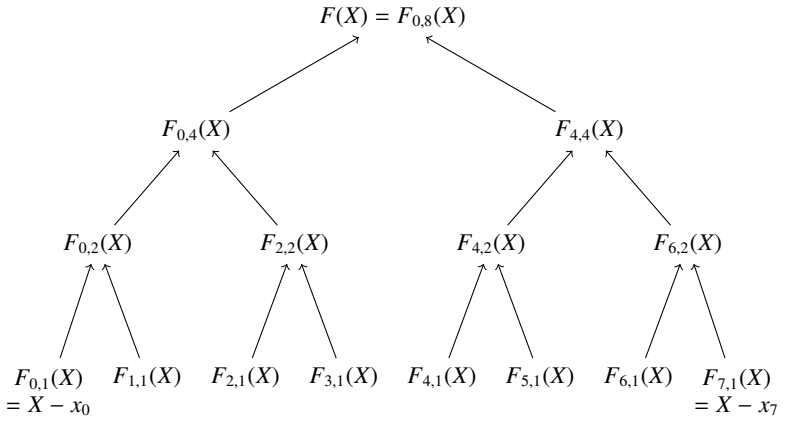

Figure 1.1 Example of a product tree to compute a polynomial $F(X)$ of degree 8 from its roots $x_0, \ldots, x_7$.

since if any $\gcd(x_\sigma - x_\tau, n)$ is non-trivial, so will be $\gcd(h, n)$.

The idea of the FFT stage two is to compute this product via fast algorithms for polynomial multi-point evaluation with complexity in $O(d(\log d)^2)$ ring operations, rather than evaluating all $O(d^2)$ terms of (1.1) individually.

1.2.1 The product tree algorithm

Let $F(X)$ be the polynomial whose roots are the x_τ, i.e., $F(X) = \prod_{\tau \in T} (X - x_\tau)$, and similarly $G(X) = \prod_{\sigma \in S} (X - x_\sigma)$. Both $F(X)$ and $G(X)$ can be computed in $O(M(d) \log d)$ operations — still assuming $d = \lfloor B_2^{1/2} \rfloor$ — over $\mathbb{Z}/n\mathbb{Z}$ with the "product tree" algorithm and fast polynomial multiplication [10], where $M(d) = M_n(d)$ denotes the complexity of multiplying polynomials of degree d over $\mathbb{Z}/n\mathbb{Z}$. The product tree algorithm works by recursively multiplying polynomials of equal degree ℓ, starting with the linear polynomials $X - x_\tau$. Assuming $d = 2^k$ is a power of 2, there are k recursion levels, each costing $O(M(\ell)d/\ell)$, for the stated total cost $O(M(d)k) = O(M(d) \log d)$. If d is not a power of 2, a variant of the product tree can be used that multiplies polynomials of degrees differing by 1 where necessary.

Algorithm 1 shows how to compute the product tree when d is a power of 2, and Figure 1.1 shows an example for a polynomial of degree 8.

Algorithm 1 Product tree algorithm

Inputs: d, a power of 2. $r_0, \ldots, r_{d-1} \in \mathbb{Z}/n\mathbb{Z}$.
Output: $F(X) \in \mathbb{Z}/n\mathbb{Z}[X]$ of degree d with roots r_i, $0 \le i < d$.
for $i \leftarrow 0, d-1$ **do**
  $F_{i,1}(X) = X - r_i$ ▷ Initialise leaves
end for
for $\ell \leftarrow 1, 2, 4, \ldots, d/2$ **do** ▷ Powers of 2
  for $i \leftarrow 0, 2\ell, 4\ell, \ldots, d-2\ell$ **do**
    $F_{i,2\ell}(X) \leftarrow F_{i,\ell}(X) F_{i+\ell,\ell}(X)$
  end for
end for
return $F_{0,d}(X)$

1.2.2 The POLYEVAL algorithm

Given $F(X) = \prod_{\tau \in T}(X - x_\tau)$, we can rewrite the product (1.1) on page 3 as $h = \prod_{\sigma \in S} F(x_\sigma) \bmod n$. To evaluate $F(X)$ on many points x_σ efficiently, Montgomery suggests the "POLYEVAL" algorithm.

We first note that $F(x_\sigma) = F(X) \bmod (X - x_\sigma)$; this is a degree-0 polynomial, i.e., an element of $\mathbb{Z}/n\mathbb{Z}$. Let $F(X), G(X) \in (\mathbb{Z}/n\mathbb{Z})[X]$ be polynomials as defined in Section 1.2.1, both assumed to have degree $d = 2^k$. Let $G_{i,\ell}(X) \in (\mathbb{Z}/n\mathbb{Z})[X]$, $\ell \mid d$, $\ell \mid i$, $0 \le i < d$, be the nodes of the product tree associated with $G(X)$. We can evaluate all $F(x_\sigma)$ in total time $O(M(d) \log d)$ by recursively reducing $F(X)$ modulo the various $G_{i,\ell}(X)$, forming a "remainder tree", whose leaf nodes then contain the values $F(X) \bmod (X - x_\sigma) = F(x_\sigma)$.

This process uses the nodes $G_{i,\ell}(X)$ from the product tree for $G(X)$ from the root downwards to the leaves, i.e., in the opposite direction of how it was created. Thus it is not practical to create the $G_{i,\ell}(X)$ on-the-fly. To obtain acceptable performance it seems necessary to compute the product tree for $G(X)$ ahead of time and store its nodes, resulting in memory requirements of $O(d \log d)$ residues. Montgomery [14, 3.7] points out that the product tree can be kept on external storage; the GMP-ECM implementation offers this option, reducing main memory use to only $O(d)$ ring elements while typically incurring only negligible delay for reading coefficients of $G_{i,\ell}(X)$ from storage during POLYEVAL.

Once all the values $F(x_\sigma)$ are computed, we take their product $h = \prod_{\sigma \in S} F(x_\sigma)$ and compute $g = \gcd(h, n)$. If for any $p \mid n$, $\sigma \in S$, and $\tau \in T$ we have $x_\sigma \equiv x_\tau \pmod p$, then $h \equiv 0 \pmod p$ and thus $p \mid g$, revealing the factor p.

Polynomial modular reduction

To compute a polynomial modular reduction efficiently, Montgomery proposes an algorithm RECIP that computes a reciprocal of a polynomial in the following sense: let $G(X) \in (\mathbb{Z}/n\mathbb{Z})[X]$ be a polynomial of degree d whose leading coefficient is invertible in $\mathbb{Z}/n\mathbb{Z}$, then $\mathrm{RECIP}(G) = \left\lfloor \frac{X^{2d}}{G(X)} \right\rfloor$, where $\left\lfloor \frac{f(X)}{g(X)} \right\rfloor$ for polynomials $f(X)$, $g(X)$ is the polynomial quotient with remainder discarded.

The algorithm proceeds by applying the familiar Newton iteration for reciprocals on the Laurent series $G(X)/X^d$, see [14, 3.5] for details. Given $\mathrm{RECIP}(G)$, we can compute $F(X) \bmod G(X)$ by first computing a quotient

$$H(X) = \left\lfloor \frac{F(X)}{G(X)} \right\rfloor = \left\lfloor \frac{\lfloor F(X)/X^d \rfloor \mathrm{RECIP}(G)}{X^d} \right\rfloor, \tag{1.2}$$

then

$$F(X) \bmod G(X) = F(X) - G(X)H(X). \tag{1.3}$$

(This can be seen as a translation for polynomials of Barrett's algorithm to reduce integers.) We note that, if $G_{i,2\ell}(X) = G_{i,\ell}(X)G_{i+\ell,\ell}(X)$ and the polynomials $G_{i,\ell}$ are monic, then

$$\mathrm{RECIP}(G_{i,\ell}) = \lfloor \lfloor \mathrm{RECIP}(G_{i,2\ell})X^{-\ell} \rfloor G_{i+\ell,\ell}(X)X^{-\ell} \rfloor$$

and similarly for $\mathrm{RECIP}(G_{i+\ell,\ell})$. The POLYEVAL algorithm uses this to compute the required polynomial reciprocals $\mathrm{RECIP}(G_{i,\ell})$ and $\mathrm{RECIP}(G_{i+\ell,\ell})$ from $\mathrm{RECIP}(G_{i,2\ell})$ as we traverse the remainder tree, so that the more expensive Newton iteration to compute a reciprocal is required only at the root node.

Algorithm 2 on page 8 gives pseudo-code for POLYEVAL when d is a power of 2, and Figure 1.2 on page 7 presents an example of a remainder tree for a polynomial of degree 8. As for the product tree algorithm, the algorithm can be adjusted to handle slightly unbalanced trees if d is not a power of 2.

1.2.3 The POLYGCD algorithm

Montgomery also suggested the "POLYGCD variant", in which h is interpreted as the resultant $\mathrm{Res}(F, G)$, whose computation reduces to a polynomial GCD. This can also be computed in $O(M(d) \log d)$ operations. We note that the concept of a polynomial GCD over $\mathbb{Z}/n\mathbb{Z}$ is not well-defined when n is composite, as Montgomery's example

$$\gcd(X^2 + 9X + 8, 2X + 9) \pmod{35}$$

shows. Applying the Euclidean algorithm leads to the problem of dividing $2X + 9$ by 14, but 14 is not invertible in $\mathbb{Z}/35\mathbb{Z}$. Fortunately this works to

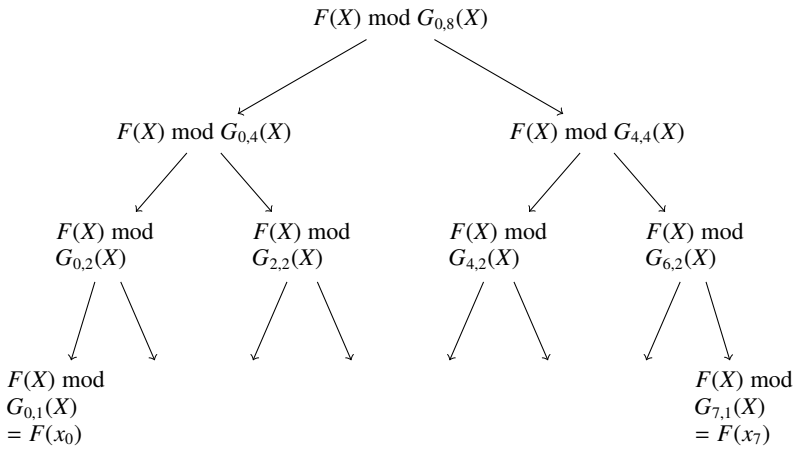

Figure 1.2 Example of a remainder tree to evaluate a polynomial $F(X)$ on 8 points $x_0, \ldots, x_7$ which form the roots of the polynomial $G(X)$. The polynomials $G_{i,\ell}(X)$ are the nodes in the product tree for building $G(X)$ from its roots.

our advantage, since the fact that 14 is not invertible means that $\gcd(14, 35)$ gives a factor of 35.

Montgomery's POLYGCD algorithm is an optimized variant of the subquadratic algorithm for polynomial GCDs presented in [1, 8.9]. The algorithm is based on two ideas: the GCD can be computed quickly from the two polynomials' quotient sequence (defined below), and computing the quotient of two polynomials of nearly equal degrees requires only a few of their highest coefficients.

Given two polynomials $f(X)$, $g(X)$ over a field K, $\deg(f) \geq \deg(g)$, define their remainder sequence as $r_0(X) = f(X)$, $r_1(X) = g(X)$, $r_i(X) = r_{i-2}(X) \bmod r_{i-1}(X)$ for $2 \leq i \leq k$, with $r_k(X) = 0$ so that $r_{k-1}(X) = \gcd(f(X), g(X))$, and their quotient sequence as $q_i(X) = \lfloor r_{i-2}(X)/r_{i-1}(X) \rfloor$ for $2 \leq i \leq k$. Thus $r_i(X) = r_{i-2}(X) - q_i(X) r_{i-1}(X)$, and for $0 \leq i \leq j$

$$\begin{pmatrix} r_j(X) \\ r_{j+1}(X) \end{pmatrix} = A_{i,j} \begin{pmatrix} r_i(X) \\ r_{i+1}(X) \end{pmatrix} \tag{1.4}$$

with

$$A_{i,j} = \prod_{\ell=i}^{j} \begin{pmatrix} 0 & 1 \\ 1 & -q_\ell(X) \end{pmatrix}. \tag{1.5}$$

From the quotient sequence we can evaluate the product in (1.5) quickly with a

Algorithm 2 POLYEVAL algorithm

Inputs: d, a power of 2. $F(X) \in (\mathbb{Z}/n\mathbb{Z})[X]$, a monic polynomial of degree d. $G_{i,\ell}(X) \in (\mathbb{Z}/n\mathbb{Z})[x]$, monic polynomials as computed by Algorithm 1 on page 5 from the roots $x_0, \ldots, x_{d-1}$.
Output: $F(x_0), \ldots, F(x_{d-1})$.
$H_{0,d}(X) \leftarrow F(X) - G_{0,d}(X)$ ▷ Equivalent to $F(X) \bmod G_{0,d}(X)$
for $\ell \leftarrow d/2, \ldots, 2, 1$ **do** ▷ Powers of 2
  for $i \leftarrow 0, 2\ell, 4\ell, \ldots, d - 2\ell$ **do**
    ▷ Compute reciprocals needed for modular reduction
    $\mathrm{RECIP}(G_{i,\ell}) \leftarrow \lfloor \lfloor \mathrm{RECIP}(G_{i,2\ell}) X^{-\ell} \rfloor G_{i+\ell,\ell}(X) X^{-\ell} \rfloor$
    $\mathrm{RECIP}(G_{i+\ell,\ell}) \leftarrow \lfloor \lfloor \mathrm{RECIP}(G_{i,2\ell}) X^{-\ell} \rfloor G_{i,\ell}(X) X^{-\ell} \rfloor$
    ▷ $\mathrm{RECIP}(G_{i,2\ell})$ is not needed any more after this
    $H_{i,\ell}(X) \leftarrow H_{i,2\ell}(X) \bmod G_{i,\ell}(X)$ ▷ Using (1.2), (1.3) on page 6
    $H_{i+\ell,\ell}(X) \leftarrow H_{i,2\ell}(X) \bmod G_{i+\ell,\ell}(X)$
    ▷ $H_{i,2\ell}(X)$ is not needed any more after this
  end for
end for
return $H_{0,1}(X), \ldots, H_{d-1,1}(X)$ ▷ Equals $F(x_0), \ldots, F(x_{d-1})$

product tree algorithm similar to Algorithm 1 on page 5, but operating on 2×2 matrices of polynomials, and obtain the desired GCD with the matrix-vector product (1.4).

The quotient of two polynomials $f(X)$, $g(X)$ can be computed from their high-order coefficients $\lfloor f(X)/X^i \rfloor$ and $\lfloor g(X)/X^i \rfloor$ if $i \leq 2\deg(g) - \deg(f)$ [1, Lemma 8.6]. We note that in the remainder sequence of two polynomials, successive terms typically have degrees differing by 1. This allows computing the quotient sequence without generating all the $O(\deg(f)^2)$ coefficients of the full remainder sequence, a necessary criterion for a sub-quadratic GCD algorithm.

The POLYGCD algorithm uses a HGCD sub-routine which recursively computes approximately the first half of the quotient sequence for two polynomials $r_i(X)$, $r_{i+1}(X)$ and collects the corresponding product matrix $A_{i,j}$ (with $j \approx (k + i)/2$ as in (1.5), k being the length of the remainder sequence. With this matrix, the corresponding $r_j(X)$, $r_{j+1}(X)$ are computed via (1.4) and HGCD is used again. This process iterates until $r_{k-1}(X) = \gcd(f(X), g(X))$ is reached.

Montgomery's PhD dissertation includes the algorithm and a proof of its correctness [14, 3.8], analysis of its running time [14, 3.9], and proves that if the algorithm is applied to polynomials defined over a ring $\mathbb{Z}/n\mathbb{Z}$ which is not a field, it will either compute the correct GCD or discover a proper factor of n.

With the current state of the art, the POLYEVAL variant seems faster (by

a small constant factor), but requires more space – $\Theta(d \log d)$ coefficients in $\mathbb{Z}/n\mathbb{Z}$, instead of only $O(d)$ for POLYGCD, although the latter has a relatively large proportionality constant. When storing the product tree of $G(X)$ on disk, POLYEVAL uses both less main memory and less CPU time, usually making it the preferred choice.

For ways of performing fast polynomial multiplication in $(\mathbb{Z}/n\mathbb{Z})[X]$, we refer to [19]. Montgomery [14] suggested performing several FFTs modulo small primes – chosen so that finding a suitable primitive root of unity as needed for the FFT is easy – and then recovering the coefficients by the Chinese Remainder Theorem. Another approach is to use the "Kronecker-Schönhage trick" [8, pg. 44], which encodes polynomials as integers.

1.2.4 Choice of points of evaluation

Stage two discovers a factor p if, for some $\sigma \in S$ and $\tau \in T$, $x_\sigma = x_\tau \pmod p$. The x-coordinate of a point and its inverse are identical; thus the factor is found if $\sigma Q = \pm\tau Q$ on $E_{a,b} \bmod p$, or $(\sigma \pm \tau)Q = O_E \bmod p$. Let $o = \mathrm{ord}(Q)$ on $E_{a,b} \bmod p$, then stage two finds the factor if $\sigma \pm \tau \equiv 0 \pmod o$. We assume that o has no small prime factors, as $Q = kP$ where k is the product of primes and prime powers $\leq B_1$. We also assume o is prime; if it is composite, then it is very likely too large to be found. Thus it is sufficient to choose $T = \{j : 0 < j < d/2, \gcd(j, d) = 1\}$, assuming even d, as all primes in $(B_1, B_2]$ can then be written as either $\sigma - \tau$ or $\sigma + \tau$ with $\sigma \in S$, $\tau \in T$.[3] This way, the set T has only half the size, reducing the degree of the polynomial $F(X)$ accordingly.

Using powers

In the context of the $p - 1$ and $p + 1$ methods, Montgomery [13] suggested computing (using our present notation) not σQ and τQ, but $\sigma^2 Q$ and $\tau^2 Q$, producing a match if $(\sigma^2 - \tau^2) \equiv 0 \pmod o$, so that these algorithms likewise need only a set T of half the size, even though in their respective group, an element and its inverse do not have an identical residue. The scalar values have approximately twice the bit-length, so the cost of computing multiples of Q approximately doubles, but this is easily offset by having to compare only half as many pairs for a match.

The first author and Suyama extended the idea to allowing higher powers than 2. When using $\sigma^k Q$ and $\tau^k Q$, $k \geq 1$, in the context of ECM, a factor p is found if $\sigma^k - \tau^k \equiv 0 \pmod o$ or $\sigma^k + \tau^k \equiv 0 \pmod o$. With prime o, this is equivalent to $(\frac{\sigma}{\tau})^{2k} - 1 \equiv 0 \pmod o$.

[3] To hit all primes up to B_2, the set S should now contain all multiples of d up to $B_2 + d/2$.

The polynomial $X^{2k} - 1$ factors into a total of $\nu(2k)$ cyclotomic polynomials, where $\nu(2k)$ is the number of divisors of $2k$; the two linear factors generate the primes in $(B_1, B_2]$, but the polynomial factors of higher degree may be divisible by primes larger than B_2. A naïve analysis estimates the probability that a (not too large) prime $o > B_2$ occurs as a divisor of some $\sigma^k - \tau^k$ or $\sigma^k + \tau^k$ as $1 - (1 - (\nu(2k) - 2)/o)^{|S||T|}$.

The polynomial $X^k - 1 \pmod{o}$, o prime, $o \neq k$, has exactly $\gcd(o - 1, k)$ roots. This leads to a "clustering" effect for the primes greater than B_2 generated by the Brent-Suyama extension using k-th powers: for primes congruent to $1 \pmod{2k}$, we have $2k$ roots, causing such primes to occur very frequently as divisors of some $\sigma^{2k} - \tau^{2k}$. For primes congruent to $-1 \pmod{2k}$, we have only 2 roots, belonging to the two linear cyclotomic factors which generate the primes up to (and perhaps a few slightly larger than) B_2, and therefore using $\sigma^k Q, \tau^k Q$ instead of $\sigma Q, \tau Q$ does not help for such primes. Thus, primes o where $\gcd(o-1, k)$ is large are over-represented, while primes where this GCD is small are under-represented.

Using Dickson polynomials

In his PhD dissertation, Montgomery investigated which other functions can be used in place of powers and found that Dickson polynomials $D_{k,\alpha}(X)$ of degree k have favourable properties. Like powers, $D_{k,\alpha}(X) - D_{k,\alpha}(Y)$ and $D_{k,\alpha}(X) + D_{k,\alpha}(Y)$ have a total of $\nu(2k)$ irreducible polynomial factors between them. The advantage of using Dickson polynomials for stage two is that $D_{k,\alpha} - z \equiv 0 \pmod{o}$ may have $\gcd(o - 1, k)$ or $\gcd(o + 1, k)$ roots, depending on the quadratic character of $z^2 - 4\alpha^k$. As σ and τ range over S and T, respectively, both cases occur roughly equally often, giving an average number of roots of $(\gcd(o - 1, k) + \gcd(o + 1, k))/2$. This largely avoids the aforementioned clustering effect.

Montgomery [14, Chap. 5] gives a detailed analysis of both powers and Dickson polynomials for stage two of ECM. He also describes how to compute the required multiples of Q efficiently via finite differences tables.

As an example of the efficacy of the various choices, we give here the expected number of curves required to find a factor p of about 60 decimal digits, as computed by GMP-ECM, which takes into account the effect of the Brent-Suyama extension with powers or Dickson polynomials. The recommended parameters for finding 60-digit factors are $B_1 = 2.6 \cdot 10^8$, $B_2 \approx 3.18 \cdot 10^{12}$, and using Dickson polynomials of degree 30.

Polynomial	X^1	X^{30}	$D_{30,-1}(X)$
Expected number of curves	54038	48508	47888

As can be seen, the expected number of curves is 12.8% larger when not using the Brent-Suyama extension (case X^1) than with default parameters. When using the power X^{30} instead of the Dickson polynomial, the expected number of curves is only 1.3% larger, but as using powers or Dickson polynomials of equal degree has identical computational cost, the latter are always preferable.

1.2.5 A numerical example

We give a numerical example to illustrate the effectiveness of ECM with an FFT extension for stage two.

The Mersenne number $2^{1163} - 1$ has "small" factors $p_{12} = 848181715001$, $p_{21} = 337097300570078978047$, and the quotient $N = (2^{1163} - 1)/(p_{12}p_{21})$ is a 318-digit composite. In April 2010, Joppe Bos, Thorsten Kleinjung, Arjen Lenstra and Peter Montgomery factored N using ECM with an FFT extension for stage two, using about $5 \cdot 10^4$ curves with $B_1 = 3 \cdot 10^9$, $B_2 = 10^{14}$ (stage one was performed on a Playstation 3 [5]). They found that $N = p_{73} \cdot p_{246}$, where

$$p_{73} = 104281604294184575004295220668008979441501466832985039303191048352645648 7$$

is a 73-digit prime, and $p_{246} = 4201 \cdots 5263$ is a 246-digit prime. The lucky elliptic curve E was defined by Suyama's parameter $\sigma = 3000085158$, and had group order

$$g = 2^2 \cdot 3^2 \cdot 5 \cdot 23 \cdot 1429 \cdot 28229 \cdot 139133 \cdot 249677 \cdot 389749 \cdot 15487861 \cdot 47501591 \cdot 111707179 \cdot g_2 \cdot g_1,$$

where the two largest prime factors of g are $g_1 = 13007798103359$ and $g_2 = 431421191$. Note that $g_1/B_1 \approx 4336$.

1.3 FFT extension for the $p - 1$ and $p + 1$ methods

The $p-1$ and $p+1$ methods can be seen as a "special case" of the ECM method, where computations are done in the multiplicative ring of integers modulo n instead of in the group defined by a pseudo-random elliptic curve. These methods work well when n has a factor p such that $p - 1$ (or $p + 1$, respectively) is smooth. The $p - 1$ method was invented by Pollard in 1974 [17], and the $p + 1$ method by Williams in 1982 [18].[4] As for ECM, those methods admit

[4] The $p + 1$ method was discovered independently by the first author of this chapter in 1981 and used to find factors of Mersenne numbers, e.g., the 21-digit factor $p = 122551752733003055543$ of $2^{439} - 1$, see [6]. Here $p - 1 = 2.439.139580583978363389$

a stage two which reduces to computing values Q^σ and Q^τ in the corresponding (multiplicatively written) group, where Q was computed in stage one, with σ, τ being taken from two sets S and T, respectively, and checking for a match modulo p between all pairs of Q^σ and Q^τ by using fast polynomial arithmetic.

One main difference with ECM is that, if say S is comprised of integers in an arithmetic progression, then for $p-1$ the values $Q^\sigma \bmod n$ form a geometric progression modulo n. In that case algorithms exist for computing $G(X) = \prod_{\sigma\in S} (X - Q^\sigma)$ in $O(M(d))$ operations instead of $O(M(d)\log d)$ as in the generic case. In [15] Montgomery and the second author describe how those algorithms can be tuned for $p-1$ and $p+1$. We give a brief overview below for the $p-1$ case, and we refer the reader to [15] for the $p+1$ case. This FFT continuation was already suggested by Pollard [17], and using polynomial evaluation along a geometric progression was implemented by Montgomery and Silverman [16].

Assume n has a factor p such that $p-1$ is divisible by all primes up to the stage one limit B_1, except a prime factor $h = \sigma+\tau$ where $\sigma \in S$ and $\tau \in T$. Then $Q^h = 1 \bmod p$, thus $Q^\sigma = Q^{-\tau} \bmod p$. If we compute $F(X) = \prod_{\tau\in T} (X - Q^{-\tau})$ and $G(X) = \prod_{\sigma\in S} (X - Q^\sigma)$, then the resultant of F and G will yield (up to sign)

$$h = \prod_{\sigma\in S}\prod_{\tau\in T} (Q^\sigma - Q^{-\tau});$$

thus h will be divisible by p, and $\gcd(h, n)$ will reveal p.

We describe here a simplified version of the algorithm in [15]. It chooses a highly composite integer $P > 2$, and an even convolution length ℓ with $0 < \phi(P) < \ell$. For best performance, ℓ should be roughly twice as large as $\phi(P)$.

We choose T to be a set of representatives of $(\mathbb{Z}/P\mathbb{Z})^*$ which we can factor into a sum of sets, where each set is an arithmetic progression of prime length, centered at zero. We use the identities $(\mathbb{Z}/(mn)\mathbb{Z})^* = n(\mathbb{Z}/m\mathbb{Z})^* + m(\mathbb{Z}/n\mathbb{Z})^*$ if $\gcd(m, n) = 1$ and $(\mathbb{Z}/p^k\mathbb{Z})^* = (\mathbb{Z}/p\mathbb{Z})^* + \sum_{i=1}^{k-1} p^i(\mathbb{Z}/p\mathbb{Z})$ for prime p. An arithmetic progression $R_n = \{2i - n - 1 : 1 \le i \le n\}$ can be factored into a sum of arithmetic progressions of prime length via $R_{mn} = R_m + mR_n$. We use R_{p-1} as the set of representatives for $(\mathbb{Z}/p\mathbb{Z})^*$ for odd p.

As a small example one might take $P = 105$ were we use the factorization $35\{-1, 1\} + 21\{-3, -1, 1, 3\} + 15\{-5, -3, -1, 1, 3, 5\}$ as representatives of $(\mathbb{Z}/105\mathbb{Z})^*$. We can further factor it into the sum of sets

$$T = 35\{-1, 1\} + 42\{-1, 1\} + 21\{-1, 1\} + 45\{-1, 1\} + 15\{-2, 0, 2\}, \qquad (1.6)$$

is not nearly as smooth as $p + 1 = 2^3.3.19.4673.13171.36037.121169$, so the $p-1$ method is less effective than the $p+1$ method.

each of prime length. This factorization of representatives of $(\mathbb{Z}/P\mathbb{Z})^*$ into a sum of small sets is used to build the polynomial $F(X)$ by linear recursion, which is therefore faster than the more general product tree computed by Algorithm 1 on page 5. We will usually refer to T in fully factored representation, i.e., as a sum of arithmetic progressions of prime length. A reciprocal Laurent polynomial (RLP) $f(X)$ of degree $2d$ is a power series of the form $\sum_{i=0}^{d} f_i(X + X^{-1})$ with $f_d \neq 0$. The elements in T are used to generate the roots of a reciprocal Laurent polynomial $F(X)$ which will therefore have degree $t = \phi(P)$. This polynomial will be evaluated along a geometric progression. These two steps are described below.

1.3.1 Constructing $F(X)$ by scaling and multiplying

The reason why we require T to be symmetric around zero is that the $p + 1$ method constructs an element of $\mathbb{Z}/n\mathbb{Z}$ of the form $R = Q + Q^{-1}$ where Q is not explicitly known and may be in a quadratic extension of $\mathbb{Z}/n\mathbb{Z}$. The use of RLPs allows performing all arithmetic in stage two of the $p + 1$ method referencing only R, and never Q, by the use of Chebyshev polynomials $V_k(X) \in \mathbb{Z}[X]$ which satisfy $V_k(X + 1/X) = X^k + 1/X^k$, see [15].

With $P > 2$, $t = \phi(P)$ is even, so it is possible to include an arithmetic progression of length 2 in the set T. The RLP $F(X)$ is constructed such that its roots are Q^τ for $\tau \in T$, i.e., using the fact that T is symmetric around 0,

$$\begin{aligned} F(X) &= \prod_{\tau \in T, \tau > 0} ((X - Q^\tau)(X - Q^{-\tau})/X) \\ &= \prod_{\tau \in T, \tau > 0} (X + X^{-1} - V_\tau(R)). \end{aligned}$$

Each term in the product is an RLP, thus $F(X)$ is an RLP, so only $t/2 + 1$ of its coefficients need to be stored.

For the purpose of this exposition, we will assume the $p-1$ method where Q is known, as this simplifies the algorithm for constructing $F(X)$. It can be found in [15] how to construct $F(X)$ in a way that works for both the $p - 1$ and $p + 1$ methods.

We start the recursive construction of $F(X)$ with one of the summands of cardinality 2 in T, in our example with $\{-35, 35\}$, which ensures we obtain an RLP of degree 2 (where the degree of an RLP is the maximal degree difference between its monomials):

$$F_1(X) = X + X^{-1} - V_{35}(R).$$

Next, we process all summands of odd cardinality in T, in our example $\{-30, 0, 30\}$,

i.e., from $F_1(X)$ we construct the RLP

$$F_2(X) = \prod_{\tau\in\{-35,35\}+\{-30,0,30\},\tau>0} X + X^{-1} - V_\tau(R)$$

of degree 6 by scaling $F_1(X)$ and multiplying:

$$F_2(X) = F_1(Q^{-30}X)F_1(X)F_1(Q^{30}X).$$

The summands of odd cardinality in T are processed early as they are more difficult than the cardinality 2 case because they require products of RLPs of unequal degrees, so we prefer to handle them while the degrees of the RLPs involved are small.

Finally, the remaining summands of cardinality 2 in T are processed. Each one doubles the degree of the resulting RLP. If the i-th summand that we process is $\{-k, k\}$, we compute

$$F_i(X) = F_{i-1}(Q^{-k}X)F_{i-1}(Q^kX);$$

with m summands in T (in our example, $m = 5$), $F_m(X) = F(X)$, see Figure 1.3 on page 15.

As P is chosen to be highly composite, $\phi(P)$ and thus t contain many factors of 2 so that the cost of building $F(X)$ is dominated by processing sets of cardinality 2. The total cost is therefore in $O(M(d/2) + M(d/4) + M(d/8) + \ldots + M(1)) = O(M(d))$, smaller by a factor of order $\log d$ than the cost of the generic product tree in Algorithm 1 on page 5.

For the $p+1$ method, Q is not explicitly known, but the arithmetic for building $F(X)$ by scaling and multiplication can be reformulated entirely in terms of Chebyshev polynomials which reference only $R = Q + Q^{-1}$. Multiplication of RLPs can be performed efficiently via weighted FFTs, see [15] for details.

1.3.2 Evaluation of a polynomial along a geometric progression

After $F(X)$ is built from the roots Q^τ, $\tau \in T$, we evaluate it along the geometric progression Q^σ, $\sigma \in S$, with $S = \{kP : k \in \mathbb{N}, B_1 \le kP \le B_2\}$. Note that this choice of S may not cover quite all the primes in $(B_1, B_2]$, depending on the set T, see [15, §5].

A polynomial can be evaluated along a geometric progression with Bluestein's algorithm [4], related to the chirp-z transform which is a generalization of the length-ℓ DFT to points of evaluation along a geometric progression other than powers of an ℓ-th primitive root of unity ω_ℓ.

For a polynomial $F(X)$ of degree t with coefficient vector in monomial basis,

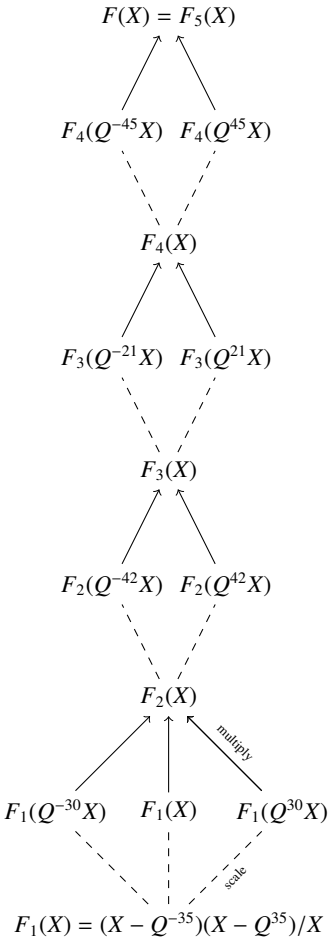

Figure 1.3 Example of computing a reciprocal Laurent polynomial $F(X)$ of degree 24 by scaling and multiplying. Each scaling/multiplication step corresponds to one summand in the set $T = \{-35, 35\} + \{-42, 42\} + \{-21, 21\} + \{-45, 45\} + \{-30, 0, 30\}$.

the length-ℓ DFT of the coefficient vector is

$$\begin{aligned} \bar{f_k} &= \sum_{0 \le i \le t} f_i \omega_\ell^{ik} \qquad \text{for } 0 \le k < \ell \\ &= F(\omega_\ell^k), \end{aligned}$$

i.e., it evaluates the polynomial along the geometric progression ω_ℓ^k for $0 \le$

$k < \ell$. The chirp-z transform allows us to change these points of evaluation to other geometric progressions.

For simplicity, we show how to evaluate $F(Q^{kP})$ for successive values of $k = 0, 1, \ldots, s-1$ with Bluestein's algorithm. We write

$$\begin{aligned} F(Q^{kP}) &= \sum_{0 \le i \le t} f_i Q^{ikP} \\ &= \sum_{0 \le i \le t} f_i Q^{ikP} Q^{(i^2+k^2)P/2} Q^{-(i^2+k^2)P/2} \\ &= Q^{k^2P/2} \sum_{0 \le i \le t} f_i Q^{i^2P/2} Q^{-(i^2-2ik+k^2)P/2} \\ &= Q^{k^2P/2} \sum_{0 \le i \le t} f_i Q^{i^2P/2} Q^{-(i-k)^2P/2} \\ &= Q^{k^2P/2} \tilde{F}(X) W(X)[k] \qquad \text{for } 0 \le k < s \end{aligned}$$

where $\tilde{F}(X)$ is a polynomial of degree t with coefficients $\tilde{f_i} = f_i Q^{i^2P/2}$, $0 \le i \le t$, and $W(X)$ is a power series with coefficients $w_i = Q^{-i^2P/2}$ for $-s < i \le t$. The desired values of $F(Q^{kP})$ (up to the factors $Q^{k^2P/2}$, which are harmless for our purpose but could be divided out) for $0 \le k < s$ occupy the coefficients $0, \ldots, s-1$ in the product; these are not affected by wrap-around if a cyclic convolution of length at least $s + t$ is used. The method is readily adapted to sets S which do not start at 0.

This demonstrates how a multi-point evaluation of a polynomial of degree t along a geometric progression of length s can be effected by a single cyclic convolution product of length $\ell = s + t$; assuming $s \approx t$, this has complexity in $O(M(t))$, faster by a factor of order $\log(t)$ than the general multi-point evaluation in Algorithm 2 on page 8.

The bound B_2 and thus the number of points of evaluation s can be chosen so that ℓ is a power of 2, allowing efficient use of FFT-based algorithms for the cyclic convolution.

For the $p + 1$ method, where the value of Q is not explicitly known, some of the arithmetic needs to be performed in a quadratic extension of $\mathbb{Z}/n\mathbb{Z}$, increasing computational cost and memory requirements. More technical details, including memory allocation savings, can be found in [15]. This algorithm is implemented in GMP-ECM both for $p-1$ and $p+1$. Examples of large factors found using this FFT extension are the 55-digit factor

$$p = 4090528046283359170676873346935001420047744892805041229$$

of $81901^{41} - 1$ found by A. Reich in March 2015, where the largest factor of

$p-1$ is the 15-digit prime 804102884120257; and the 60-digit factor

$q=725516237739635905037132916171116034279215026146021770250523$

of the Lucas number L_{2366}, found by Montgomery and the second author in October 2007, with largest factor of $q+1$ the 15-digit prime 483576618980159.

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974. (Cited on pages 7 and 8.)

[2] D. J. Bernstein, P. Birkner, T. Lange, and C. Peters. ECM using Edwards curves. *Mathematics of Computation*, 82(282):1139–1179, 2013. (Cited on page 2.)

[3] I. F. Blake, G. Seroussi, and N. P. Smart, editors. *Advances in Elliptic Curve Cryptography*. Cambridge University Press, 2005. (Not cited.)

[4] L. Bluestein. A linear filtering approach to the computation of discrete Fourier transform. *IEEE Transactions on Audio and Electroacoustics*, 18(4):451–455, 1970. (Cited on page 14.)

[5] J. W. Bos, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery. Efficient SIMD arithmetic modulo a Mersenne number. In E. Antelo, D. Hough, and P. Ienne, editors, *IEEE Symposium on Computer Arithmetic – ARITH-20*, pages 213–221. IEEE Computer Society, 2011. (Cited on page 11.)

[6] R. P. Brent. New factors of Mersenne numbers (preliminary report), II. *AMS Abstracts*, 3:132, 82T–10–22, 1982. (Cited on page 11.)

[7] R. P. Brent. Some integer factorization algorithms using elliptic curves. *Australian Computer Science Communications*, 8:149–163, 1986. (Cited on page 1.)

[8] R. P. Brent and P. Zimmermann. *Modern Computer Arithmetic*. Cambridge University Press, 2010. (Cited on page 9.)

[9] H. M. Edwards. A normal form for elliptic curves. *Bulletin of the American Mathematical Society*, 44:393–422, July 2007. (Cited on page 2.)

[10] J. Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, Cambridge, 1999. `http://www-math.uni-paderborn.de/mca`. (Cited on page 4.)

[11] D. S. Johnson, T. Nishizeki, A. Nozaki, and H. S. Wilf. *Discrete algorithms and complexity*. Academic Press, Boston, 1987. (Not cited.)

[12] H. W. Lenstra Jr. Factoring integers with elliptic curves. *Annals of Mathematics*, 126(3):649–673, 1987. (Cited on page 1.)

[13] P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987. (Cited on pages 1 and 9.)

[14] P. L. Montgomery. *An FFT extension of the elliptic curve method of factorization*. PhD thesis, University of California, 1992. (Cited on pages 1, 2, 5, 6, 8, 9, and 10.)

[15] P. L. Montgomery and A. Kruppa. Improved stage 2 to p±1 factoring algorithms. In A. J. van der Poorten and A. Stein, editors, *Algorithmic Number Theory – ANTS-VIII*, volume 5011 of *Lecture Notes in Computer Science*, pages 180–195. Springer, 2008. (Cited on pages 1, 12, 13, 14, and 16.)

[16] P. L. Montgomery and R. D. Silverman. An FFT extension to the $p - 1$ factoring algorithm. *Mathematics of Computation*, 54(190):839–854, 1990. (Cited on pages 1, 2, and 12.)

[17] J. M. Pollard. Theorems on factorization and primality testing. *Proceedings of the Cambridge Philosophical Society*, 76:521–528, 1974. (Cited on pages 1, 2, 11, and 12.)

[18] H. C. Williams. A $p + 1$ method of factoring. *Mathematics of Computation*, 39(159):225–234, 1982. (Cited on pages 3 and 11.)

[19] P. Zimmermann and B. Dodson. 20 years of ECM. In F. Hess, S. Pauli, and M. E. Pohst, editors, *Algorithmic Number Theory – ANTS-VII*, volume 4076 of *Lecture Notes in Computer Science*, pages 525–542. Springer-Verlag. Erratum: `http://www.loria.fr/~zimmerma/papers/`, 2006. (Cited on pages 1, 2, 3, and 9.)