A New Polynomial Factorization Algorithm and its Implementation*

Victor Shoup
Bellcore, 445 South St., Morristown, NJ 07960
shoup@bellcore.com

Abstract

We consider the problem of factoring univariate polynomials over a finite field. We demonstrate that the new *baby step/giant step factoring method*, recently developed by Kaltofen & Shoup, can be made into a very practical algorithm. We describe an implementation of this algorithm, and present the results of empirical tests comparing this new algorithm with others. When factoring polynomials modulo large primes, the algorithm allows much larger polynomials to be factored using a reasonable amount of time and space than was previously possible. For example, this new software has been used to factor a "generic" polynomial of degree 2048 modulo a 2048-bit prime in under 12 days on a Sun SPARC-station 10, using 68 MB of main memory.

1 Introduction

We consider the problem of factoring a univariate polynomial of degree n over the field $\mathbf{F}_p$ of p elements, where p is prime.

This problem has been well-studied, and many algorithms for its solution have been proposed. In general, the running time of any algorithm for this problem depends on n and p, and the best choice of algorithm will depend on the relative sizes of these two parameters. In this paper, we shall concentrate on the case where p is large; for concreteness, say $\log_2 p = \Omega(n)$. This case arises, for example, in algorithms for counting points on elliptic curves over $\mathbf{F}_p$ [13]. In the case where p is very small, especially the extreme case where $p = 2$, the algorithmic issues are quite different from those in the case we are considering here.

We demonstrate that the new *baby step/giant step factoring method*, recently developed by Kaltofen & Shoup [11], can be made into a very practical algorithm. We describe an implementation of this algorithm, and present the results of empirical tests comparing this new algorithm with others. When factoring polynomials modulo large primes, the algorithm allows much larger polynomials to be factored using a reasonable amount of time and space than was previously possible. For example, this new software has been used to factor a "generic" polynomial of degree 2048 modulo a 2048-bit prime in under 12 days on a Sun SPARC-station 10, using 68 MB of main memory.

1.1 Comparison of factoring methods

To measure the running time of factoring algorithms, we will count the number of scalar operations (arithmetic operations and zero tests in $\mathbf{F}_p$). In all of the known factoring methods, a key operation

*To appear in *Jornal of Symbolic Computation*, 1996.

is multiplication of polynomials over $\mathbf{F}_p$. We shall assume that multiplication of two degree d polynomials uses $O(M(d))$ scalar operations, where $M(d) = d \log d \operatorname{loglog} d$. This running-time bound is attained using the Fast Fourier Transform (FFT) [1], and (as we shall later see) is quite realistic in practice.

To measure the space complexity of factoring algorithms, we will count the number of scalars (elements of $\mathbf{F}_p$) that need to be stored.

Berlekamp's *null-space method* [2] reduces the factoring problem to that of finding elements in the null space of a certain linear map defined on a vector space of dimension n over $\mathbf{F}_p$. Using standard elimination techniques, it can be implemented so as to use

$$O(n^3 + M(n) \log n \log p)$$

scalar operations, and space for $O(n^2)$ scalars. The exponent 3 in the running time can be reduced using asymptotically fast (but generally impractical) matrix techniques.

Cantor and Zassenhaus's *degree-separation method* [5] works by first partially factoring the polynomial so as to separate irreducible factors of differing degree, and then completing the factorization (if necessary) by separating irreducible factors of the same degree. It can be implemented so as to use $O(nM(n) \log p)$ scalar operations and space for $O(n)$ scalars. For the large p case, this method is in theory and in practice much slower than the null-space method; however, it is uses much less space.

Von zur Gathen and Shoup introduced a *fast conjugation technique* [21] to speed up the degree-separation method. This technique yields new and asymptotically fast algorithms for computing successive pth powers in polynomial quotient rings over $\mathbf{F}_p$. With the fast conjugation technique, the degree-separation method can be implemented so as to use

$$O(nM(n) \log n + M(n) \log n \log p)$$

scalar operations and space for $O(n^{1.5})$ scalars. For large p, this algorithm is asymptotically the fastest known. However, it appears to be quite impractical, and our experience indicates that it will be slower than the null-space method for n up to at least several thousands.

Kaltofen and Lobo introduced a *black-box variant* of the null-space method [9] by applying Wiedemann's linear system solving techniques [22] to the null-space method. The black-box variant can be implemented in a variety of ways, achieving a variety of simultaneous time/space bounds (including those of the above three methods).

Kaltofen and Shoup introduced a *baby step/giant step technique* [11] that can be applied to either the degree-separation method or the black-box variant of the null-space method. This new method yields a factoring algorithm that for small p is asymptotically the fastest known, using $O(n^{1.82} \log p)$ scalar operations. This technique can also be used to obtain an algorithm—appropriate for large p—that uses

$$O(n^{2.5} + M(n) \log n \log p)$$

scalar operations and space for $O(n^{1.5})$ scalars. Our main goal is to demonstrate that this new algorithm is quite practical, in that it runs faster than the null-space method for reasonably small n, and—perhaps more significantly—requires much less space than the null-space method.

Table 1: Empirical Results (SPARC-ELC)

n	New Algorithm		Null-Space Method		Maple
64	$13''$	1 MB	$13''$	1 MB	$13'25''$
128	$1'52''$	1 MB	$1'51''$	2 MB	$25^h13'$ –
256	$18'10''$	2 MB	$25'31''$	6 MB	
512	$3^h25'$ –	4 MB	$7^h59'$ –	48 MB	
1024	$50^h58'$ –	11 MB			

Table 2: Empirical Results (SPARC-10)

n	New Algorithm		Null-Space Method	
64	$5''$	1 MB	$5''$	1 MB
128	$46''$	1 MB	$45''$	2 MB
256	$6'52''$	2 MB	$9'19''$	6 MB
512	$1^h17'$ –	4 MB	$2^h46'$ –	48 MB
1024	$16^h45'$ –	18 MB		
2048	$272^h09'$ –	68 MB		

1.2 Empirical results

We briefly summarize here our empirical results; more details are to be found later in the paper.

Von zur Gathen [20] has suggested a "polynomial factorization challenge" consisting of a family of benchmarks for polynomial factorization algorithms. For each n, von zur Gathen's nth benchmark consists of a degree n polynomial modulo an $(n+2)$-bit prime. The benchmarks are chosen so that they are easy to describe and to generate, but yet appear to behave as "generic" inputs.

In response to von zur Gathen's challenge, Monagan [16] implemented the null-space method in Maple on a DEC station 3100. The largest of von zur Gathen's benchmarks reported to be factored in [16] was the $n = 200$ benchmark, which took approximately 27 hours. In contrast, on a Sun SPARC-station ELC, whose performance is roughly similar to that of the DEC station 3100, we factored this same polynomial in 8.3 *minutes* with the baby step/giant step method, and 10.2 minutes using our implementation of the null-space method.

Also, the MAGMA computer algebra system was used to factor the $n = 300$ benchmark [3]. This was done using an implementation of the null-space method on a Sun MP670, which is somewhat faster than our SPARC-station ELC. The total running time was 110 hours. On our SPARC-station ELC, the running time for the baby step/giant step method on this input was 48 *minutes*, and for our implementation of the null-space method, 70 minutes.

We propose in this paper a new set of benchmarks that is in the same spirit as von zur Gathen's, but we correct what we believe is a slight technical deficiency. Our benchmark number n consists of a degree n polynomial modulo an n-bit prime. We ran our implementations of the baby step/giant step method and of the null-space method on several of these benchmarks, using a Sun SPARC-station ELC. The running-time and space requirements are shown in Table 1. Running times are reported as $hours^h minutes' seconds''$. Space is reported in terms of megabytes (MB) of main memory. For $n = 1024$, the space requirement of the null-space method exceeded that available on any of our machines—in our implementation, the space requirement would exceed 300 MB. For comparison, we also include in Table 1 the running time of the built-in factorization routine in Maple for $n = 64$ and $n = 128$. That routine uses the degree-separation method.

At a late stage in this project, we obtained access to a Sun SPARC-station 10, which is between 2 and 3 times faster than the ELC. This machine also has more memory (100 MB) than any of the ELC workstations we had available (16–64 MB). Using this machine, we obtained corresponding data, which we report in Table 2. It was only on this machine that the $n = 2048$ benchmark was factored. The discrepancy between the space used in the baby step/giant step method for $n = 1024$ in the two data sets is due to a different time/space trade-off that was made.

In implementing the new factoring algorithm, we have developed a very portable software library for polynomial arithmetic which utilizes fast methods for polynomial multiplication. This software has proven very useful in other contexts. For example, this software has been used to count the number of points over $\mathbf{F}_p$ where p is a 914-bit (375 decimal digit) prime in a week's time on a network of workstations, which represents the state of the art for this problem. These results are reported in [13].

1.3 Overview of algorithm

We now present an overview of the new algorithm. While it is possible to apply the baby step/giant step technique to either the null-space or the degree-separation methods, we work only with the degree-separation method, as our experience indicates that this is a slightly more practical approach. The degree-separation method has three basic steps:

square-free factorization Partially factor a given polynomial into a product of square-free factors.

distinct-degree factorization Partially factor a given square-free polynomial into a product of factors, where each factor is the product of irreducibles all of the same degree (and this degree is also part of the output).

equal-degree factorization Given a square-free polynomial whose irreducible factors all have the same degree, along with that degree, factor it into irreducibles.

Using an algorithm of Yun [23], the square-free factorization stage can be accomplished essentially in the time to compute a GCD. Using the recursive "Half-GCD" method (see [1, Chapter 8]), this takes $O(M(n)\log n)$ scalar operations. The space requirement is $O(n)$ scalars. We do not discuss this any further in this paper.

Using the new baby step/giant step technique, distinct-degree factorization can be implemented so as to use

$$O(n^{2.5} + M(n)\log p)$$

scalar operations and space for $O(n^{1.5})$ scalars, where the implied constants in both the space and time estimates are quite small, making this quite practical. This method is described in §2.

For the equal-degree factorization step, we apply the *fast trace computation technique* of [21]; this is discussed in §3. This stage of the algorithm uses in the worst case

$$O(n^2 \log n + M(n)\log n \log p)$$

scalar operations and space for $O(n^{1.5})$ scalars. Again, the implied constants here are quite small. Note that for a random input, the number of irreducible factors at this stage will likely be 1, in which case no work is required, or 2, in which case the above running-time estimate drops to

$$O(n^2 \log n + \log p).$$

The other sections of this paper are organized as follows.

In §4 we present an algorithm for computing minimum polynomials over $\mathbf{F}_p$ of elements in polynomial quotient rings over $\mathbf{F}_p$. This problem appears as a sub-task in factoring algorithms, but our results here may be of independent interest as well.

In §5, we discuss the particular variant of the null-space method that we implemented.

In §§6 and 7 we discuss the algorithms that were employed for arithmetic in $\mathbf{F}_p$ and $\mathbf{F}_p[x]$. Our algorithms for polynomial arithmetic use the FFT, and are much faster than the "classical" methods. In §8, we discuss some of the implementation details of long-integer arithmetic and related problems. We certainly do not claim that all of the techniques in §§6-8 are original. Similar work has also been done by others (see Montgomery [17] in the context of integer factorization, and Morain [18] in the context of counting points on elliptic curves).

In §9, we discuss the results of timing experiments with our software, including a precise description of our factoring benchmarks.

In §10, we make some concluding remarks, including a brief discussion of the application of the baby step/giant step technique to the black-box variant of the null-space method, as well as the use of classical polynomial arithmetic.

2 Distinct-degree factorization

The motivation for the new distinct-degree factorization algorithm is the following fact: for any nonnegative integers a and b, the polynomial $x^{p^a} - x^{p^b} \in \mathbf{F}_p[x]$ is divisible precisely by those irreducible polynomials in $\mathbf{F}_p[x]$ whose degree divides $a - b$. This fact follows, for example, from [15, Theorem 3.20].

Here is a high-level description of the new algorithm. The details of how each step is to be implemented are deferred until later. Let $f \in \mathbf{F}_p[x]$ be the polynomial (assumed square-free) to be factored, with $n = \deg(f)$. The output of the algorithm is a list $f_1, \ldots, f_n$ of polynomials in $\mathbf{F}_p[x]$, where for $1 \le d \le n$, f_d is the product of the degree d irreducible factors of f.

Let $B = \lfloor n/2 \rfloor$, $\ell = \lfloor \sqrt{B} \rfloor$, $m = \lceil B/\ell \rceil$, and $h_0 = x \in \mathbf{F}_p[x]$.

Step 1 Compute $h_1 = x^p \bmod f \in \mathbf{F}_p[x]$ via repeated squaring.

Step 2 For $2 \le i \le \ell$, compute $h_i = x^{p^i} \bmod f \in \mathbf{F}_p[x]$.

Step 3 For $1 \le j \le m$, compute $H_j = x^{p^{\ell j}} \bmod f \in \mathbf{F}_p[x]$.

Step 4 For $1 \le j \le m$, compute

$$I_j = \prod_{0 \le i \le \ell - 1} (H_j - h_i) \bmod f \in \mathbf{F}_p[x].$$

Step 5 Initialize $f_1, \ldots, f_n$ to 1, and then do the following:

$f^* \leftarrow f$
for $j \leftarrow 1$ to m do
  $g \leftarrow \gcd(f^*, I_j)$; $f^* \leftarrow f^*/g$
  for $i \leftarrow \ell - 1$ down to 0 do
    $f_{\ell j - i} \leftarrow \gcd(g, H_j - h_i)$; $g \leftarrow g/f_{\ell j - i}$
if $f^* \neq 1$ then $f_{\deg(f^*)} \leftarrow f^*$

That concludes the high-level description of the new distinct-degree factorization algorithm. The correctness of this algorithm follows easily from the fact that for $1 \leq j \leq m$, $\gcd(f, I_j)$ is equal to the product of those irreducible factors of f whose degree divides an integer lying in the interval $[j\ell - (\ell - 1), j\ell]$.

2.1 Modular Composition

Using the fast conjugation technique, it is possible to show that the new distinct-degree factorization algorithm can be implemented so as to use

$$O(nM(n)\log n + M(n)\log p)$$

scalar operations and space for $O(n^{3/2})$ scalars. However, this algorithm seems hopelessly impractical at the moment.

A more practical approach is the following. In Step 2, we can compute $h_2, \ldots, h_\ell$ iteratively, computing h_{i+1} as $h_i(h_1) \bmod f$. In Step 3, we can similarly compute $H_1, \ldots, H_m$ iteratively, by setting $H_1 = h_\ell$, and computing H_{j+1} as $H_j(H_1) \bmod f$.

Thus, to perform Steps 2 and 3, we have to solve $\approx \sqrt{2n}$ instances of the *modular-composition problem*: given polynomials g, h, and f in $\mathbf{F}_p[x]$, where f has degree n and g and h are of degree less than n, compute $g(h) \bmod f$. Moreover, note that f and h remain fixed throughout many instances of the problem, which we can use to our advantage.

To solve the modular-composition problem, we use the method of Brent & Kung [4], which is itself a baby step/giant step technique. We first choose a parameter t $(1 \leq t \leq n)$, and build a table of powers $h^i \bmod f$ for $0 \leq i \leq t$ using $\approx t$ multiplications by h modulo f. Then we express g as

$$g(x) = \sum_{0 \leq j < n/t} g_j(x) y^j,$$

where $\deg(g_j(x)) < t$ and $y = x^t$. Any one $g_j(x)$ can be evaluated at $h \bmod f$ by accessing the power-table and performing $\approx nt$ scalar multiplications and additions. Then, to evaluate g at h modulo f, we perform a degree $\approx n/t$ Horner evaluation which takes $\approx n/t$ multiplications by h^t modulo f.

In total, this method requires

- $\approx n^2$ scalar multiplications and additions;
- $\approx t + n/t$ multiplications modulo f;

- space for $\approx tn$ scalars.

To optimize the time for a single modular composition, it is clear that t should be chosen near $n^{1/2}$.

If many modular compositions are performed with the same f and the same argument h, as in the new distinct-degree factorization algorithm, we compute the power-table once using $\approx t$ multiplications modulo f, and then each individual modular composition requires just $\approx n^2$ scalar multiplications and additions, plus $\approx n/t$ multiplications mod f. Thus, to optimize the running time in this case, one would choose t larger than $n^{1/2}$. However, the space requirement grows with t; if space should become a problem, and if polynomial multiplication is reasonably fast (as it is in our implementation), one can scale back the parameter t without degrading the time performance significantly.

Note that more generally if we are computing $g(h) \bmod f$, and we allow g to have arbitrary degree $m-1$, we can utilize the same method as described above using $\approx mn$ scalar multiplications and additions, $\approx t + m/t$ multiplications mod f, and space for $\approx tn$ scalars. Here, t is a parameter with $1 \leq t \leq m$.

2.2 Saving space

To reduce main-memory requirements of the distinct-degree factorization algorithm, we proceed as follows. In Step 1, we compute the h_i's one at a time as described above, and write them to disk. This avoids keeping all of the h_i's and the power table for the modular compositions in main memory at the same time.

We do the same thing in computing the H_j's in Step 3.

Steps 4 and 5 are interleaved, so that I_j is computed at the beginning of the jth iteration in the outer loop of Step 5.

At the beginning of combined Step 4/5, we read in all of the h_i's from disk. Each H_j needs to be read in once when it is needed.

Thus, if t is the table-size parameter in the modular-composition algorithm in Steps 2 and 3, the main-memory space requirement will be $\approx n \cdot \max(t, \sqrt{n/2})$ scalars. Disk space is needed for $\approx n\sqrt{2n}$ scalars, and each such element is written to disk once, and read from disk once.

In Steps 2 and 3, the default size for the parameter t in our implementation is $2\lfloor n^{1/2} \rfloor$. This was overridden in our experiments when space became a serious concern: in the $n = 1024$ case in Table 1, where we set $t = 25$, and in the $n = 2048$ case in Table 2, where we set $t = 64$.

2.3 Reducing the number of GCD computations

In Step 5, we need to compute many GCD's (most of which will yield 1). We can reduce the time spent in this step by reducing the total number of GCD computations.

To do this, we employ the following "blocking" technique. For example, in the outer loop in Step 5, we compute the product P of I_1 through I_b mod f^*, where b is a parameter, and we then compute $\gcd(P, f^*)$. If this is not 1, we then go back and take individual GCD's. Then we compute the product of the next block of b I_j's, and so on.

If b is not too large, most of the computations $\gcd(P, f^*)$ will yield 1, so we will effectively trade b (expensive) GCD computations for 1 GCD plus b (cheap) polynomial multiplications mod f^*. This technique requires space for an extra $\approx bn$ scalars, since we have to buffer the I_j's. A similar blocking technique is used in the inner loop of Step 5. In our experiments, we found that by choosing $b = 4$, the time spent computing GCD's was reduced to a very small fraction of the total running time.

3 Equal-Degree Factorization

We assume that we are given $f \in \mathbf{F}_p[x]$ of degree n, and an integer d, where f is the product of $r = n/d$ distinct irreducibles each of degree d. We also assume we are given $x^p \bmod f$ (this is computed already in the distinct-degree factorization algorithm). We want to completely factor f into irreducibles.

For $k \geq 0$, let $T_k \in \mathbf{F}_p[x]$ be defined as

$$T_k = \sum_{0 \leq i < k} x^{p^i}.$$

Our algorithm consists of two steps.

Trace Computation Choose a random polynomial $g \in \mathbf{F}_p[x]$ of degree less than n, and compute $h = T_d(g) \bmod f$.

Factor Extraction Using h computed above, compute a factorization of f.

The output from the factor extraction step may not be a complete factorization; that depends on the choice of g in the trace computation step. However, with probability at least $1 - r^2/p$, the factorization will be complete; if it is not (which we can detect trivially, knowing d and r), we can repeat the algorithm. For the large values of p we are mainly considering in this paper, this is not a serious issue.

We discuss the Trace Computation step next, and the Factor Extraction step in §3.2.

3.1 Trace Computation

In [21], a *fast trace computation technique* was introduced. It was observed that for nonnegative integers a and b,

$$T_{a+b} = T_a(x^{p^b}) + T_b.$$

This leads to a repeated-squaring type of algorithm for computing $T_k(g) \bmod f$ using $O(\log k)$ modular compositions. Algorithm 5.2 in [21] is such an algorithm. We describe here an algorithm that is slightly more efficient.

The inputs are the integer k, and the polynomials g, f, and $(x^p \bmod f)$. Let $k = (b_\ell \dots b_0)_2$ be the binary representation of k. When the following algorithm terminates, $w = T_k(g) \bmod f$.

$u \leftarrow x^p \bmod f$; $v \leftarrow g$; $w \leftarrow 0$
for $i \leftarrow 0$ to ℓ do
    if $b_i = 0$ then
        $(u, v, w) \leftarrow (u(u) \bmod f, (v(u) \bmod f) + v, w)$
    else
        $(u, v, w) \leftarrow (u(u) \bmod f, (v(u) \bmod f) + v, (w(u) \bmod f) + v)$

The correctness follows easily by induction: at the beginning of each loop iteration, we have

$$\begin{aligned} u &= x^{p^{2^i}} \bmod f, \\ v &= T_{2^i}(g) \bmod f, \\ w &= T_{k_i}(g) \bmod f, \qquad \text{where } k_i = (b_{i-1} \dots b_0)_2. \end{aligned}$$

Each iteration of the loop, we perform either 2 or 3 compositions mod f, where all compositions have the same argument u. This is convenient, as then we need to build just one power table per loop iteration, cutting down significantly on the running time.

In comparison, Algorithm 5.2 in [21], computes (per loop iteration) either 2 compositions with the same argument, or 4 compositions with two different arguments. As modular composition is quite expensive, our algorithm leads to a significant savings.

3.2 Factor Extraction

We assume we are given a polynomial $f \in \mathbf{F}_p[x]$ of degree n, and an integer d such that f is square-free with $r = n/d$ irreducible factors each of degree d. We also assume we are given $h = T_d(g) \bmod f$ for randomly chosen $g \in \mathbf{F}_p[x]$, with $\deg(g) < n$. We also explicitly assume at this point that $p > 2$.

Suppose $f = f_1 \dots f_r$ is the factorization of f into irreducibles. Then by the Chinese Remainder Theorem, we have an isomorphism of $\mathbf{F}_p$-algebras

$$\begin{aligned} \mathbf{F}_p[x]/(f) &\cong \bigoplus_{i=1}^{r} \mathbf{F}_p[x]/(f_i) \\ &\cong \bigoplus_{i=1}^{r} \mathbf{F}_{p^d} \end{aligned}$$

that maps $a \in \mathbf{F}_p$ onto $(a, a, \dots, a)$.

Let B be the sub-algebra of $\mathbf{F}_p[x]/(f)$ that maps onto $\bigoplus_{i=1}^{r} \mathbf{F}_p$ via this isomorphism. The only property of h that we will use is that it is a random element in B. That is, h maps onto an element

$$(a_1, \dots, a_r) \in \bigoplus_{i=1}^{r} \mathbf{F}_{p^d},$$

where each "component" a_i of h is a random element of $\mathbf{F}_p$. With probability at least $1 - r^2/p$, these components will be mutually distinct.

Assume for now that these components are indeed distinct. We can proceed in one of two different ways to factor f.

The simple extraction method In this method, we select $b \in \mathbf{F}_p$ at random, and compute

$$\gcd((h + b)^{(p-1)/2} - 1, f),$$

which can be done using $O(M(n)(\log p + \log n))$ scalar operations. We expect (for random b) that this will split f into two factors, each with roughly the same number of factors. We can then recursively apply this technique on the two factors.

It follows that the expected total cost of the recursion is

$$O(M(n)\log r(\log p + \log n))$$

scalar operations (see, e.g., [21, Lemma 4.1] for a detailed analysis of this type of recursion).

The minimum-polynomial extraction method This method avoids the costly computation of a $(p-1)/2$-th power modulo f, and for small r (which is expected on random inputs) is significantly faster than the simple extraction method.

This method proceeds as follows.

First, we compute the minimum polynomial $h^* \in \mathbf{F}_p[x]$ of h modulo f; that is, we compute the monic polynomial h^* of least degree such that $h^*(h) \equiv 0 \pmod{f}$. One can easily see that

$$h^* = \prod_{i=1}^{r}(x - a_i).$$

We can obtain h^* at essentially the same cost as composing a degree $2r$ polynomial with h mod f, namely $O(r^{1/2}M(n) + rn)$ scalar operations and space for $O(r^{1/2}n)$ scalars; this is explained in §4.

Having f, h, and h^* as prescribed, we proceed recursively as follows. We choose $b \in \mathbf{F}_p$ at random and compute

$$h_1^* = \gcd((x+b)^{(p-1)/2} - 1, h^*),$$

and $h_2^* = h^*/h_1^*$. This takes $O(M(r)(\log p + \log r))$ scalar operations. We expect (for random b) that this splits h^* into two factors h_1^* and h_2^* of roughly equal degree.

Next we compute $f_1 = \gcd(h_1^*(h), f)$, and $f_2 = f/f_1$. This takes $O(r^{1/2}M(n) + rn + M(n)\log n)$ scalar operations, and it splits f into two factors f_1 and f_2, of degrees $d \cdot \deg(h_1^*)$ and $d \cdot \deg(h_2^*)$ respectively.

Now we recursively apply this method twice with

$$f \leftarrow f_1,\ h \leftarrow (h \bmod f_1),\ h^* \leftarrow h_1^*,$$

and

$$f \leftarrow f_2,\ h \leftarrow (h \bmod f_2),\ h^* \leftarrow h_2^*.$$

It follows (see again [21, Lemma 4.1]) that the expected total cost of the minimum-polynomial extraction method is

$$O(\log r(M(r)\log p + M(n)(r^{1/2} + \log n) + rn))$$

scalar operations and space for $O(r^{1/2}n)$ scalars.

It is clear that for small r, which is typical for random inputs, the minimum-polynomial extraction method is much faster than the simple extraction method of factor extraction. Indeed, when $r = 2$, the minimum-polynomial method takes $O(\log p + M(n)\log n)$ scalar operations, whereas the simple extraction method takes $O(M(n)(\log p + \log n))$ scalar operations.

We have been assuming that the polynomial h computed in the trace computation step has r distinct components $a_1, \ldots, a_r$. If this is not the case, either of the above methods for factor extraction can be easily modified so as to detect this situation, and to return just a partial factorization of f. We omit the details of this.

4 Computing Minimum Polynomials

In this section, we consider the following problem. Let $f, h \in \mathbf{F}_p[x]$ with $n = \deg(f) > \deg(h)$. The problem is to compute the minimum polynomial h^* of h modulo f, i.e., the nonzero, monic polynomial h^* of least degree such that $h^*(h) \equiv 0 \pmod{f}$. We assume also that we are given an upper bound m on the degree of h^*; if not, one can always choose the trivial upper bound $m = n$.

This problem arose in §3, but is perhaps of more general interest as well.

An obvious approach is to compute the powers $h^i \bmod f$, for $0 \le i \le m$, and find a linear relation using Gaussian elimination. For large m this is quite costly, requiring $O(mM(n) + m^2 n)$ scalar operations and space for $O(nm)$ scalars.

An asymptotically faster and more space-efficient approach is described in Shoup [19], using $O(m^{1/2}M(n) + mn)$ scalar operations and space for $O(m^{1/2}n)$ scalars. However, the techniques used there prove only the existence of an algorithm, and do not give an explicit algorithm. We give one here.

The algorithm has two basic steps.

Power Projection Choose a random $\mathbf{F}_p$-linear map $R : \mathbf{F}_p[x]/(f) \to \mathbf{F}_p$. For $i \ge 0$, define $u_i = R(h^i \bmod f)$. Compute u_i for $0 \le i < 2m$.

Berlekamp-Massey Using the output from the previous step, compute the minimal polynomial of the sequence $\{u_i\}_{i \ge 0}$, which is a linearly-generated sequence whose minimal polynomial divides h^*. This is the well-known "Berlekamp-Massey problem" from coding theory.

In the power-projection step, the random map R is chosen by simply choosing random elements r_i from $\mathbf{F}_p$ for $0 \le i < n$, and defining $R(x^i \bmod f) = r_i$.

In the Berlekamp-Massey step, the output polynomial will depend on the random choices made in the first step. The "success" probability that the algorithm correctly outputs the polynomial h^* can be bounded from below in two different ways: this probability is at least $1 - m/p$ [10], which is a useful bound when p is large with respect to m; for small p, this probability is bounded from below by a constant times $1/\lceil \log_p m \rceil$ [22].

The Berlekamp-Massey step can be carried out with essentially a GCD computation, taking $O(M(m) \log m)$ scalar operations [8]. If the resulting polynomial has degree less than m, we can test if it actually annihilates h using the modular-composition algorithm.

4.1 Power Projection

It remains to discuss the implementation of the power projection step. Let us consider the general problem of computing $u_i = R(h^i \bmod f)$ for $0 < i < k$, for an arbitrary k, and for an arbitrary linear projection R. The obvious way is simply to compute successive powers of $h \bmod f$, and project them using R. This takes $\approx k$ multiplications mod f, and $\approx nk$ scalar multiplications and additions.

We shall describe describe a much faster power projection algorithm that uses $\approx 2\sqrt{k}$ multiplications mod f, $\approx nk$ scalar multiplications and additions, and space for $\approx n\sqrt{k}$ scalars. But first, we make the following observation.

Consider the $k \times n$ matrix representing the $\mathbf{F}_p$-linear map

$$R \mapsto (R(h^i \bmod f))_{i=0}^{k-1}.$$

The transpose of this matrix represents the linear map

$$g \mapsto g(h) \bmod f,$$

where $g \in \mathbf{F}_p[x]$ has degree less than k. This is just the modular-composition problem. By the "transposition principle" [12], under certain technical restrictions, an algorithm for modular composition can be transformed into one with the same time complexity for power projection. Thus, the power projection problem might also be called the transposed modular-composition problem.

The transposition principle is very useful for proving the existence of algorithms, but actually coming up with an explicit, practical algorithm requires a bit more effort. We begin by defining a sub-task employed by our power projection algorithm, which we call *transposed modular multiplication.*

For an arbitrary polynomial $g \in \mathbf{F}_p[x]$ of degree less than n, let $M[\![g]\!]$ be the $\mathbf{F}_p$-linear map on $\mathbf{F}_p[x]/(f)$ that for $u \in \mathbf{F}_p[x]$ sends $(u \bmod f)$ to $(gu \bmod f)$. Consider an arbitrary $\mathbf{F}_p$-linear map $Q : \mathbf{F}_p[x]/(f) \to \mathbf{F}_p$, represented as $(q_0, \ldots, q_{n-1})$, where $q_i = Q(x^i \bmod f)$ for $0 \leq i < n$. The sub-task is: given f, g, and Q as above, compute $Q \circ M[\![g]\!]$, i.e., compute $(q_0', \ldots, q_{n-1}')$ such that $q_i' = Q(g \cdot x^i \bmod f)$ for $0 \leq i < n$.

Consider the matrix representing the map $Q \mapsto Q \circ M[\![g]\!]$, with respect to the power basis modulo f. Then the transpose of this matrix is the matrix of the linear map $M[\![g]\!]$. Again by the transposition principle, one can perform a transposed modular multiplication as fast as one can perform a modular multiplication. In §7.5 we give an explicit algorithm for the transposed modular-multiplication problem, which theoretically and empirically is just as fast as our algorithm for modular multiplication. Thus, we assume at this point that solving one instance of this sub-task has the same cost as one multiplication mod f.

Now we can easily describe our algorithm for computing $u_i = R(h^i \bmod f)$ for $0 \leq i < k$. As in the modular-composition algorithm, we have a parameter t, $1 \leq t \leq k$, and we build a table of powers $h^i \bmod f$ for $0 \leq i \leq t$. For $j \geq 0$, we let Q_j denote $R \circ M[\![h^{jt} \bmod f]\!]$. We then do the following:

$Q_0 \leftarrow R$
for $j \leftarrow 0$ to $\lceil k/t \rceil - 1$ do
    for $i \leftarrow 0$ to $t-1$ do $u_{jt+i} \leftarrow Q_j(h^i \bmod f)$
    $Q_{j+1} \leftarrow Q_j \circ M[\![h^t \bmod f]\!]$

This algorithm uses $\approx nk$ scalar multiplications and additions in $\mathbf{F}_p$, $\approx t + k/t$ multiplications mod f, and space for tn scalars. Clearly, to optimize the time spent for a single power projection, one would choose $t \approx \sqrt{k}$.

5 Berlekamp's Null-Space Method

Since we are comparing the new factoring algorithm with Berlekamp's null-space method, we need to specify exactly which version of the null-space method we have implemented. We believe that in the case we are mainly interested in, where p is large, the version we describe here is the most efficient possible.

Assume that the input polynomial $f \in \mathbf{F}_p[x]$ has degree n and is already square-free.

The idea behind the algorithm is the following. Suppose $f = f_1 \cdots f_r$ is the factorization of f. As in §3, we have an isomorphism $\mathbf{F}_p[x]/(f) \cong \bigoplus_{i=1}^r \mathbf{F}_p[x]/(f_i)$. Let B be the sub-algebra of $\mathbf{F}_p[x]/(f)$ that maps onto $\bigoplus_{i=1}^r \mathbf{F}_p$, which has dimension r over $\mathbf{F}_p$. As in §3, our main goal is to find a random element in B. As B is the kernel of the $\mathbf{F}_p$-linear map on $\mathbf{F}_p[x]/(f)$ that sends $u \in \mathbf{F}_p[x]/(f)$ to $u^p - u$, this can be accomplished by standard techniques in linear algebra. Once we have a random element in B, we can proceed with a factor extraction procedure as in §3.

Here are the details.

Step 1 Compute $h = x^p \bmod f$ via a repeated-squaring algorithm.

Step 2 Build the $n \times n$ matrix

$$Q = (h^0 \bmod f | \cdots | h^{n-1} \bmod f) - I_n,$$

where each polynomial is expressed as a column vector whose entries are the corresponding coefficients of the polynomial, and I_n is the $n \times n$ identity matrix.

Step 3 Diagonalize Q using Gaussian elimination, producing an upper row-echelon matrix. Among other things, at this stage we discover r.

Step 4 Compute a random element in B by a back-substitution procedure using the upper row-echelon matrix from Step 3, substituting random elements in $\mathbf{F}_p$ for the free variables.

Step 5 Apply one of the factor extraction algorithms in §3.2 using the random element in B computed in Step 4.

Steps 4 and 5 may have to be repeated to get a complete factorization; however, for large p, this will happen with negligible probability.

Asymptotically, the algorithm uses in the worst case

$$O(n^3 + M(n) \log n \log p)$$

scalar operations. However, if r is small as is typical for random inputs, say $r = O(n/\log n)$, then the algorithm uses just

$$O(n^3 + M(n) \log p)$$

scalar operations, as the cost of Step 5 is then dominated by the cost of the other steps of the algorithm.

The algorithm always uses space for $\approx n^2$ scalars.

6 Arithmetic in $\mathbf{F}_p$

Elements in $\mathbf{F}_p$ are represented in our implementation as integers between 0 and $p-1$. Addition of elements in $\mathbf{F}_p$ costs one addition of $\approx \log_2 p$ bit numbers, followed by a comparison and possibly a subtraction. Subtraction in $\mathbf{F}_p$ is handled similarly.

A single multiplication in $\mathbf{F}_p$ is implemented as a multiplication of $\approx \log_2 p$ bit numbers, followed by a reduction of a $\approx 2\log_2 p$ bit number modulo a $\approx \log_2 p$ bit number.

In many situations, we need to compute a sum of products, such as $S = \sum_{i=1}^{n} a_i b_i$, where the a_i's and b_i's are elements in $\mathbf{F}_p$. In such a situation, we use the more efficient "lazy reduction" strategy: we compute each integer product, accumulate the $\approx \log_2 n + 2\log_2 p$ bit sum, and perform just one reduction mod p at the end.

This situation arises, for example, in the modular-composition algorithm (§2.1), and in our power-projection algorithm (§4.1), as well as in the classical algorithms for polynomial multiplication and division (which are used only for polynomials of very small degree, see §7).

A similar situation arises in using Gaussian elimination (in the null-space method) to transform an $n \times n$ matrix into row-echelon form. Here, we have to perform $\approx n^3/3$ scalar additions and multiplications, and $\approx n$ scalar divisions. We use the lazy reduction strategy here as well, performing Gaussian elimination as usual, but only reducing a number mod p when it is involved in either a zero test or a multiplication. This reduces number of reduction steps from $\approx n^3/3$ to $O(n^2)$, effectively cutting the running time of Gaussian elimination by about half. However, for this we pay a price: the space requirement for the matrix is effectively doubled, as we have to store n^2 integers, each with $\approx \log_2 n + 2\log_2 p$ bits, rather than $\log_2 p$ bits.

7 Arithmetic in $\mathbf{F}_p[x]$

In our implementation, classical algorithms for polynomial multiplication and division are used only for polynomials of very small degree (less than degree 30 or so). For larger polynomials, we use the following much faster FFT-based method.

Suppose that all polynomials involved in a multiplication or division have degree less than 2^M. We choose a set of "FFT-primes" $q_1, \ldots, q_\ell$, such that

- $2^{M+1} \mid q_i - 1$ for $1 \le i \le \ell$, and
- The product $P = \prod_{i=1}^{\ell} q_i$ is larger than $2^{M+1}p^2$.

To multiply two polynomials $g, h \in \mathbf{F}_p[x]$, we first reduce each coefficient (which are integers between 0 and $p-1$) modulo each of the FFT-primes primes $q_1, \ldots, q_\ell$, obtaining polynomials $g_i, h_i \in \mathbf{F}_{q_i}[x]$ for $1 \le 1 \le \ell$. Secondly, we compute $u_i = g_i \cdot h_i$ using the FFT (hence the first condition above). Finally, we apply the Chinese Remainder Theorem coefficient-wise to obtain integers in the interval $(-P/2, P/2)$, and then reduce mod p to obtain the coefficients of the product $u = g \cdot h \in \mathbf{F}_p[x]$ (hence the second condition above).

In the remainder of this section, we describe more carefully our algorithms for polynomial multiplication and related operations. We shall measure the complexity of our algorithms for polynomial arithmetic in terms of the following three basic steps.

RED-step We define one *RED-step* as the operation of reducing one integer between 0 and $p-1$ modulo each of the ℓ FFT-primes.

CRT-step We define one *CRT-step* as the operation of applying the Chinese Remainder Theorem to residues modulo $q_1, \ldots, q_\ell$, and reducing the result modulo p.

FFT-step In the standard iterative implementation of the FFT (see [6, Chapter 32]), the basic arithmetic step is the following "butterfly" operation:

$$(s, t) \leftarrow (u + v \cdot w, u - v \cdot w).$$

We define one *FFT-step* as the operation of executing one such butterfly operation for all of the ℓ FFT-primes.

Details of the choice of FFT-primes and the implementation of these three basic operations are described in §8. We remark here only that the FFT-primes are "single precision" integers, i.e., they fit in a single machine word (typically 32 bits wide).

To describe our algorithms precisely, we introduce some notation. For $0 \le k \le M+1$ and for $1 \le i \le \ell$, there exists a primitive 2^k-th root of unity in $\mathbf{F}_{q_i}$, which we denote by ω_{ik}. We assume that $\omega_{i(k-1)} = \omega_{ik}^2$ for $1 \le i \le \ell$ and $1 \le k \le M+1$. For $0 \le k \le M+1$, a 2^k-point *residue table* is a table with ℓ rows (numbered $1, \ldots, \ell$) and 2^k columns (numbered $0, \ldots, 2^k-1$), where for $1 \le i \le \ell$, the ith row consists of elements in $\mathbf{F}_{q_i}$. Addition, subtraction, and multiplication of residue tables is defined element-wise. Because these operations are computationally very inexpensive, we ignore them in our running-time analyses.

Next, we introduce some convenient operations.

RED The operation $T \leftarrow \mathrm{RED}_k(g)$ takes a polynomial $g \in \mathbf{F}_p[x]$ of degree less than 2^k, and produces a 2^k-point residue table, where the ith row of T consists of the coefficients of g (viewed as integers between 0 and $p-1$) reduced modulo q_i (padding with zeros if necessary).

If g has m nonzero coefficients, the cost of computing $\mathrm{RED}_k(g)$ is m RED-steps.

CRT The operation $g \leftarrow \mathrm{CRT}_{a\ldots b}(T)$ takes a 2^k-point residue table T, and produces a polynomial $g \in \mathbf{F}_p[x]$ of degree no more than $b-a$, where the coefficient vector of g is obtained by applying the Chinese Remainder Theorem to columns a through b of T to get a vector of length $b-a+1$ consisting of integers in the range $(-P/2, P/2)$, and then reducing these integers modulo p.

The cost of computing $\mathrm{CRT}_{a\ldots b}(T)$ is $b-a+1$ CRT-steps.

FFT The operation $S \leftarrow \mathrm{FFT}(T)$ takes a 2^k-point residue table T and produces a 2^k-point residue table S, where

$$S[i,j] = \sum_{j'=0}^{2^k-1} T[i,j'] \cdot \omega_{ik}^{jj'} \qquad (1 \le i \le \ell,\ 0 \le j < 2^k).$$

We also define the operation $T \leftarrow \mathrm{FFT}^{-1}(S)$ as the inverse operation.

The cost of computing $\mathrm{FFT}(T)$ or $\mathrm{FFT}^{-1}(S)$ is $\approx 2^{k-1}k$ FFT-steps (see [6, Chapter 32]).

With this notation, we can describe our polynomial-multiplication algorithm and its complexity quite concisely. The input consists of two polynomials $g, h \in \mathbf{F}_p[x]$ of degree at most $n-1$. Let $k = \lceil \log_2 n \rceil$. We compute the product $u = g \cdot h$ as follows:

1. $T_1 \leftarrow \mathrm{FFT}(\mathrm{RED}_{k+1}(g))$
2. $T_2 \leftarrow \mathrm{FFT}(\mathrm{RED}_{k+1}(h))$
3. $T_1 \leftarrow T_1 \cdot T_2$
4. $u \leftarrow \mathrm{CRT}_{0..2(n-1)}(\mathrm{FFT}^{-1}(T_1))$

The cost of this algorithm is

$\approx 2n$ RED-steps, $\approx 2n$ CRT-steps, and $\approx 3 \cdot 2^k k$ FFT-steps.

Squaring, of course, can be performed faster than multiplication. We can compute $u = g^2$ as follows:

1. $T \leftarrow \mathrm{FFT}(\mathrm{RED}_{k+1}(g))$
2. $T \leftarrow T \cdot T$
3. $u \leftarrow \mathrm{CRT}_{0..2(n-1)}(\mathrm{FFT}^{-1}(T))$

The cost of this algorithm is

$\approx n$ RED-steps, $\approx 2n$ CRT-steps, and $\approx 2 \cdot 2^k k$ FFT-steps.

7.1 Polynomial Division

We next consider the following problem. Given a polynomial $f \in \mathbf{F}_p[x]$ of degree n, and a polynomial $G \in \mathbf{F}_p[x]$ of degree less than $2n$, compute polynomials q and r in $\mathbf{F}_p[x]$, both of degree less than n, such that $G = qf + r$.

Using standard techniques (see [1, Chapter 8]) this can be done as follows. Let $h = x^n f(x^{-1})$ (the "reverse" of f). Let $h^* = h^{-1} \bmod x^n$, i.e., h^* consists of the first n terms of the inverse of h in the ring of formal power series $\mathbf{F}_p[[x]]$. Let $\mathrm{RECIP}(f)$ be $x^{n-1}h^*(x^{-1})$ (the "reverse" of h^*). Then we have the following fact:

$$q = \lfloor G/f \rfloor = \lfloor (G \cdot \mathrm{RECIP}(f))/x^{2n-1} \rfloor,$$

where the notation $\lfloor \cdot / \cdot \rfloor$ indicates the polynomial quotient.

This then gives us the following algorithm. The input consists of f and G as above. Let $k = \lceil \log_2 n \rceil$. We assume that we also have the following *pre-computed* values as input:

$$\begin{aligned} S_1 &= \mathrm{FFT}(\mathrm{RED}_{k+1}(\mathrm{RECIP}(f))), \\ S_2 &= \mathrm{FFT}(\mathrm{RED}_k(f \bmod (x^{2^k} - 1))). \end{aligned}$$

To compute the quotient q and remainder r, we execute the following steps:

1. $T \leftarrow \mathrm{FFT}(\mathrm{RED}_{k+1}(\lfloor G/x^n \rfloor))$
2. $T \leftarrow T \cdot S_1$
3. $q \leftarrow \mathrm{CRT}_{n-1\ldots 2(n-1)}(\mathrm{FFT}^{-1}(T))$
4. $T \leftarrow \mathrm{FFT}(\mathrm{RED}_k(q))$
5. $T \leftarrow T \cdot S_2$
6. $r \leftarrow ((G \bmod (x^{2^k} - 1)) \bmod x^n - \mathrm{CRT}_{0\ldots n-1}(\mathrm{FFT}^{-1}(T))$

Notice that in this algorithm, we are computing the remainder as

$$r = G - q \cdot f \bmod (x^{2^k} - 1),$$

which is valid, since we know that $\deg(r) < n \leq 2^k$.

The cost of this algorithm is:

$\approx 2n$ RED-steps, $\approx 2n$ CRT-steps, and $\approx 3 \cdot 2^k k$ FFT-steps.

The reason for pre-computing on S_1 and S_2 is that in many applications (including polynomial factorization algorithms), the polynomial f is held fixed for many division operations, and hence it makes sense to pre-compute the values S_1 and S_2, which depend only on f. With this pre-conditioning, the cost of division with remainder is essentially the same as that of multiplying two polynomials of degree less than n.

7.2 Computing Polynomial Inverses

To complete the algorithm in the previous subsection, we need to specify how to solve the following problem. Given a polynomial $h \in \mathbf{F}_p[x]$ of degree less than n with nonzero constant term, compute $h^{-1} \bmod x^n$, where h^{-1} is the inverse of h in the ring $\mathbf{F}_p[[x]]$ of formal power series.

Assume that $n = 2^k$ for some k (if not, pad the input). We can perform this operation using the well-known Newton iteration:

$$\begin{aligned} h_0 &= 1/(\text{constant term of } h), \\ h_i &= (2h_{i-1} - h \cdot h_{i-1}^2) \bmod x^{2^i} \qquad (i > 0). \end{aligned}$$

Here, $h_i \equiv h^{-1} \pmod{x^{2^i}}$, and so h_k is the quantity we want to compute.

The following relationship, which follows from an easy calculation, allows us to compute h_i from h_{i-1} more efficiently than a direct application of the above formulas.

Observation *For $i > 0$, we have*

$$h_i = h_{i-1} - x^{2^{i-1}}(h_{i-1} g_i \bmod x^{2^{i-1}}),$$

where

$$g_i = \lfloor \{(h_{i-1} \cdot (h \bmod x^{2^i})) \bmod (x^{2^i} - 1)\} / x^{2^{i-1}} \rfloor.$$

This leads to the following algorithm for computing h_k. First, compute $T_0 = \mathrm{RED}_k(h)$, and set $h_0 = h(0)^{-1}$. Then, execute the following steps for $i = 1, \ldots, k$.

1. $T_1 \leftarrow \mathrm{FFT}(\mathrm{RED}_i(h_{i-1}))$
2. Let T_2 be the 2^i-point residue table formed by extracting the first 2^i columns of T_0.
3. $T_2 \leftarrow T_1 \cdot \mathrm{FFT}(T_2)$
4. $g_i \leftarrow \mathrm{CRT}_{2^{i-1} \ldots 2^i - 1}(\mathrm{FFT}^{-1}(T_2))$
5. $T_2 \leftarrow \mathrm{FFT}(\mathrm{RED}_i(g_i))$

6. $T_2 \leftarrow T_1 \cdot T_2$

7. $h_i \leftarrow h_{i-1} - x^{2^{i-1}} \cdot \mathrm{CRT}_{0 \ldots 2^{i-1}-1}(\mathrm{FFT}^{-1}(T_2))$

Let us first analyze the cost of each loop iteration. To compute h_i from h_{i-1}, this comes to

$\approx 2^i$ RED-steps, $\approx 2^i$ CRT-steps, and $\approx 5 \cdot 2^{i-1} i$ FFT-steps.

This implies that the total cost of computing $h^{-1} \bmod x^n$, where $n = 2^k$, is

$\approx 3n$ RED-steps, $\approx 2n$ CRT-steps, and $\approx 5 \cdot 2^k k$ FFT-steps.

This is no more than 5/3 the cost of multiplying polynomials of degree less than n.

Note that the number of RED-steps can easily be reduced from $\approx 3n$ to $2n$, since the first 2^{i-1} terms of h_i agree with those of h_{i-1}, and the above algorithm redundantly performs the RED-step on the same coefficients many times. We did not implement this particular optimization.

Note also that in practice, one performs an iterative inversion algorithm to get the first few terms of h^{-1} before applying the Newton iteration.

7.3 Modular Squaring

Let $f \in \mathbf{F}_p[x]$ be of degree n, and $g \in \mathbf{F}_p[x]$ be of degree less than n. We want to compute $g^2 \bmod f$. This operation is of central importance and should be performed as fast as possible. For example, the cost of computing $x^p \bmod f$ using a repeated-squaring algorithm that examines the bits of p from high-order to low-order is $\approx \log_2 p$ squarings mod f, plus no more than $\log_2 p$ multiplications by $x \bmod f$ (which are essentially free). Thus, almost all of the time used to compute $x^p \bmod f$ is spent computing squares mod f.

We could simply combine the algorithms we have for squaring and division with remainder (pre-conditioned on f). This yields an algorithm for squaring mod f that uses

$\approx 3n$ RED-steps, $\approx 4n$ CRT-steps, and $\approx 5 \cdot 2^k k$ FFT-steps,

where $k = \lceil \log_2 n \rceil$.

We can in fact reduce the number of CRT-steps from $\approx 4n$ to $\approx 3n$ (effectively eliminating the application of CRT-steps to the lower-half of g^2) using the following modular-squaring algorithm. As in our division with remainder algorithm, we assume we have pre-computed the residue tables S_1 and S_2 associated with f (defined in §7.1). To compute $r = g^2 \bmod f$, we execute the following steps:

1. $T_1 \leftarrow \mathrm{FFT}(\mathrm{RED}_{k+1}(g))$

2. $T_2 \leftarrow T_1^2$

3. $v \leftarrow \mathrm{CRT}_{n \ldots 2(n-1)}(\mathrm{FFT}^{-1}(T_2))$

4. $T_2 \leftarrow \mathrm{FFT}(\mathrm{RED}_{k+1}(v))$

5. $T_2 \leftarrow T_2 \cdot S_1$

6. $q \leftarrow \mathrm{CRT}_{n-1 \ldots 2n-3}(\mathrm{FFT}^{-1}(T_2))$

7. $T_2 \leftarrow \mathrm{FFT}(\mathrm{RED}_k(q))$
8. $T_2 \leftarrow T_2 \cdot S_2$
9. Replace T_1 by the 2^k-point residue table formed by deleting the odd-numbered columns from T_1 (so $T_1 = \mathrm{FFT}(\mathrm{RED}_k(g))^2$).
10. $T_1 \leftarrow T_1 - T_2$
11. $r \leftarrow \mathrm{CRT}_{0\ldots n-1}(\mathrm{FFT}^{-1}(T_1))$

The cost of this algorithm is

$\approx 3n$ RED-steps, $\approx 3n$ CRT-steps, and $\approx 5 \cdot 2^k k$ FFT-steps.

7.4 Pre-conditioned Modular Multiplication

Let $f \in \mathbf{F}_p[x]$ be of degree n, and let $g, h \in \mathbf{F}_p[x]$ be of degree less than n. In many situations, we want to compute $r = gh \bmod f$, where not only f but also h remain fixed for many such computations. Examples of this include the construction of the matrix Q in the null-space method (§5), and our algorithm for modular composition (§2.1). Like modular squaring, this operation is of central importance, and hence should be as fast as possible.

If we just apply directly our algorithms for polynomial multiplication and division with remainder, we obtain an algorithm that uses

$\approx 4n$ RED-steps, $\approx 4n$ CRT-steps, and $\approx 6 \cdot 2^k k$ FFT-steps,

where $k = \lceil \log_2 n \rceil$.

Using a trick similar to that used in our modular-squaring algorithm, we can reduce the number of CRT-steps from $\approx 4n$ to $\approx 3n$. However, much greater savings can be attained if we pre-condition on h. More precisely, in addition to the usual residue tables S_1 and S_2 associated with f (defined in §7.1), we pre-compute:

$$\begin{aligned} S_3 &= \mathrm{FFT}(\mathrm{RED}_{k+1}(\lfloor (h \cdot \mathrm{RECIP}(f))/x^n \rfloor)), \\ S_4 &= \mathrm{FFT}(\mathrm{RED}_k(h)). \end{aligned}$$

The following algorithm performs this pre-conditioning (given S_1 and S_2):

1. $T_1 \leftarrow \mathrm{FFT}(\mathrm{RED}_{k+1}(h))$
2. $T_2 \leftarrow T_1 \cdot S_1$
3. $v \leftarrow \mathrm{CRT}_{n\ldots 2(n-1)}(\mathrm{FFT}^{-1}(T_2))$
4. $S_3 \leftarrow \mathrm{FFT}(\mathrm{RED}_{k+1}(v))$
5. Let S_4 be the 2^k-point residue table obtained by deleting the odd-numbered columns from T_1.

The cost of this pre-conditioning is

$\approx 2n$ RED-steps, $\approx n$ CRT-steps, and $\approx 3 \cdot 2^k k$ FFT-steps.

Given S_1, S_2, S_3, S_4, and g, we can compute $r = gh \bmod f$ as follows:

1. $T_1 \leftarrow \mathrm{FFT}(\mathrm{RED}_{k+1}(g))$
2. $T_2 \leftarrow T_1 \cdot S_3$
3. $q \leftarrow \mathrm{CRT}_{n-1\ldots 2n-3}(\mathrm{FFT}^{-1}(T_2))$
4. Let T_1 be the 2^k-point residue table obtained by deleting the odd-numbered columns from T_1 (so $T_1 = \mathrm{FFT}(\mathrm{RED}_k(g))$).
5. $T_2 \leftarrow \mathrm{FFT}(\mathrm{RED}_k(q))$
6. $T_1 \leftarrow T_1 \cdot S_4 - T_2 \cdot S_2$
7. $r \leftarrow \mathrm{CRT}_{0\ldots n-1}(\mathrm{FFT}^{-1}(T_1))$

The cost of this algorithm is

$\approx 2n$ RED-steps, $\approx 2n$ CRT-steps, and $\approx 3 \cdot 2^k k$ FFT-steps;

that is, about the same as the cost of a simple multiplication of two polynomials of degree less than n, and about half the cost of a multiplication mod f without pre-conditioning on h. Also note that the cost of pre-conditioning plus the cost of performing the pre-conditioned multiply is the same as that of performing a multiply mod f without pre-conditioning; therefore, it is faster to use this technique even if the same polynomial h is involved in just two multiplies mod f.

7.5 Transposed Modular Multiplication

In §4.1, the following problem arose. Let $f \in \mathbf{F}_p[x]$ be a polynomial of degree n, $h \in \mathbf{F}_p[x]$ a polynomial of degree less than n. Let A be the matrix representing the $\mathbf{F}_p$-linear map

$$M[\![h]\!] : \mathbf{F}_p[x]/(f) \to \mathbf{F}_p[x]/(f)$$

on the standard basis $1, x, \ldots, x^{n-1}$, where $M[\![h]\!]$ is the multiplication by h mod f map. The problem is to apply the transposed matrix A^T to a given vector. In our application, h remains fixed for many such applications.

In the previous subsection, we presented an algorithm for the problem of applying A to a given vector. Generally, any circuit that computes the map A can essentially be reversed to compute the transpose map A^T (see, e.g., [12]). Using this transformation technique, we have derived from our pre-conditioned modular-multiplication algorithm the following algorithm for applying A^T to a vector; we offer no other proof of correctness other than the validity of this transformation technique (and the fact that it does indeed work in practice).

Let $k = \lceil \log_2 n \rceil$. We pre-condition on h, so we assume we have the residue tables S_1, S_2 (defined in §7.1) and the residue tables S_3, S_4 (defined in §7.4). The input is taken to be the coefficient vector of a polynomial $g \in \mathbf{F}_p[x]$ of degree less than n. The output is taken to be the coefficient vector of a polynomial $u \in \mathbf{F}_p[x]$. The algorithm runs as follows:

1. $T_1 \leftarrow \mathrm{FFT}^{-1}(\mathrm{RED}_k(g))$

2. $T_2 \leftarrow T_1 \cdot S_2$

3. $v \leftarrow -\mathrm{CRT}_{0\ldots n-2}(\mathrm{FFT}(T_2))$

4. $T_2 \leftarrow \mathrm{FFT}^{-1}(\mathrm{RED}_{k+1}(x^{n-1} \cdot v))$

5. $T_2 \leftarrow T_2 \cdot S_3$

6. $T_1 \leftarrow T_1 \cdot S_4$

7. Replace T_1 by the 2^{k+1}-point residue table whose j-th column $(0 \leq j < 2^{k+1})$ is 0 if j is odd, and is column number $j/2$ of T_1 if j is even.

8. $T_2 \leftarrow T_2 + T_1$

9. $u \leftarrow \mathrm{CRT}_{0\ldots n-1}(\mathrm{FFT}(T_2))$

Just as the pre-conditioned modular-multiplication algorithm, this algorithm uses

$\approx 2n$ RED-steps, $\approx 2n$ CRT-steps, and $\approx 3 \cdot 2^k k$ FFT-steps.

7.6 Computing GCD's

For large n, we use an asymptotically fast GCD algorithm, based on the "Half-GCD" algorithm in [1, Chapter 8]. We do not present any of the details here; we only mention that in implementing this algorithm, many of the same types of optimizations can be exploited that were exploited in the previous subsections to speed the algorithm significantly.

8 Software Techniques

In this section, we describe some of the design details of the most performance-critical software in our implementation, taking into account characteristics common to many currently available architectures and compilers. All of our software is written in `C` and `C++`. One of our goals in this implementation was to write code that is highly portable, while still attaining reasonable performance.

8.1 Multi-precision arithmetic

To implement multi-precision arithmetic, we used a customized version of LIP, a `C` software library for multi-precision arithmetic written by Arjen Lenstra [14]. The *default* version of LIP is highly portable. However, by setting a flag at compile time, an *alternative* set of routines is used that is a bit less portable than the default, but on many machines is quite a bit faster. The code for the alternative version was designed by Roger Golliver, Arjen Lenstra, and this author.

For concreteness, we assume that the machine on which the software runs has 32-bit twos-complement integer arithmetic and 64-bit double-precision IEEE floating point (which has 52 explicit bits of mantissa plus 1 implicit bit). It is also assumed that integer overflow is ignored. These

Figure 1: LIP multiply with 30-bit radix

```
const double R_inv = 1.0/double(1 << 30);
const long mask = (1 << 30)-1;

inline void AddMul0(long& hi, long& lo, long a, long b, long c, long d)
{
   long t1 = a + b;
   long t2 = (a*b + t1) & mask;
   hi = long(0.25 + R_inv*(double(a)*double(b) + double(t1-t2)));
   lo = t2;
}

long AddMul(long A[], long B[], long s, int n)
{
   int i; long carry = 0;
   for (i = 0; i < n; i++)
      AddMul0(carry, A[i], B[i], s, A[i], carry);
   return carry;
}
```

assumptions are met by the vast majority of workstations and personal computers available in 1994. However, very recently, newer architectures (such as the DEC Alpha) have been introduced which support 64-bit integer arithmetic and 64-bit floating-point arithmetic. For such machines, different software techniques should be used.

In the default version of LIP, numbers are represented as vectors of type `long` (which are 32-bit integers) with radix $R = 2^{30}$; that is, the vector $A[0], A[1], \ldots, A[n-1]$, where each $A[i]$ is between 0 and $R-1$, represents the integer

$$A[0] + A[1] \cdot R + \cdots + A[n-1] \cdot R^{n-1}.$$

A fundamental problem that arises in integer multiplication and Chinese remaindering is to compute $A \leftarrow A + B \cdot s$, where A and B are multi-precision integers and s is a single-precision integer (i.e., in the range 0 through $R-1$).

The code fragment in Figure 1, written in `C++`, shows how this is accomplished in the default version of LIP.

The routine `AddMul0` computes hi and lo such that

$$hi \cdot 2^{30} + lo = a \cdot b + c + d.$$

The value lo is computed directly using one integer multiply. The value hi is computed using floating-point arithmetic; the floating-point value computed will be very close to an integer, and adding 0.25 and either rounding down or towards zero will yield the correct value of hi.

The routine `AddMul` computes $A \leftarrow A + B \cdot s$, where B has n base R digits. Any carry-out is returned as the function value.

In the alternative version of LIP, we choose the radix $R = 2^{26}$. This allows us to directly compute a 52-bit product in floating point. The only tricky part is efficiently extracting the low-order and high-order 26 bits of this product. We use the following method. First, we add the

Figure 2: LIP multiply with 26-bit radix

```
union d_or_rep { double d; long rep[2]; };
const long mask = (1 << 26)-1;
const double offset = 4503599627370496.0; // 2^52

inline void extract(long& hi, long& lo, double x)
{
   d_or_rep y;
   y.d = x;
   hi = ((y.rep[0] << 6) | (((unsigned long) y.rep[1]) >> 26)) & mask;
   lo = y.rep[1] & mask;
}

inline void add(long& hi, long& lo, long a)
{  lo = lo + a;  hi = hi + (lo >> 26);  lo = lo & mask;  }

inline void AddMul0(long& hi, long& lo, long a, long b, long c, long d)
{
   double x = double(a)*double(b) + offset;
   extract(hi, lo, x);
   add(hi, lo, c + d);
}
```

constant 2^{52} to the product. As we are assuming the IEEE floating-point standard, this has the effect of justifying the bits so that the low-order bit of the product appears as the low-order bit of the mantissa. Next, we simply extract the two words composing the floating-point number using C's `union` construct. This is inherently non-portable, but does indeed work across a wide variety of architectures and compilers (including SPARC, MIPS, IBM RS/6000, DEC VAX, and Intel 486 architectures).

Figure 2 shows the code for the alternative version of `AddMul0`. The routine `extract` returns the high-order and low-order 26-bits of the 52 explicit mantissa bits of x. The routine `add` adds the number a to the double-precision integer (hi, lo).

This version of `AddMul0` uses just one floating-point multiply, as opposed to two floating-point multiplies and one integer multiply in the default version. Depending on the "endian"-ness of the machine, the roles of `y.rep[0]` and `y.rep[1]` in `extract` may have to be reversed, which is accomplished with a compile-time flag.

Many machines (especially modern RISC processors, such as SPARC, MIPS, and RS/6000) have separate floating-point and integer units, allowing floating-point and integer operations to run in parallel. By making an explicit software pipeline that computes the next product during the same loop iteration in which the current product is being processed, a compiler with a reasonable optimizer can schedule the instructions so as to take advantage of this parallelism. Figure 3 shows the pipelined version of `AddMul`. From our experience, it is indeed necessary to code this pipeline explicitly, as most compilers are unable or unwilling to perform this optimization automatically.

The alternative version of LIP makes use of this pipelined code, and in many cases (depending on the machine and the compiler) leads to a 25%-33% reduction in running time over the unpipelined version for long-integer multiplication, and in any case never seems to run slower.

Figure 3: LIP multiply with 26-bit radix and pipelining

```
long AddMul(long A[], long B[], long s, int n)
{
   int i;  long carry = 0, hi, lo;  double x, y;
   x = double(B[0])*double(s) + offset;
   for (i = 0; i < n-1; i++) {
      y = double(B[i+1])*double(s) + offset;
      extract(hi, lo, x);
      add(carry, A[i], A[i] + carry);
      x = y;
   }
   extract(hi, lo, x);
   add(carry, A[i], A[i] + carry);
   return carry;
}
```

Figure 4: Modular multiply with 30-bit radix

```
inline long MulMod(long a, long b, long c)
{
   long q = long(double(a)*double(b)/double(c));
   long r = a*b - c*q;
   if (r < 0) r = r + c;
   else if (r >= c) r = r - c;
   return r;
}
```

8.2 Single-precision modular arithmetic

In the FFT routine used in implementing polynomial arithmetic over $\mathbf{F}_p$, a critical problem is that of computing $ab \bmod c$, where a, b, and c are single-precision integers, i.e., integers between 0 and $R-1$, where R is the multi-precision integer base.

For the default version of LIP, where $R = 2^{30}$, we use the routine in Figure 4, which first computes an approximate quotient q using floating-point, which may be off by ± 1, and then computes $r = ab - qc$ modulo 2^{32} (since we are assuming twos-complement integer arithmetic with no overflow). In our application, c remains fixed for many such operations, in which case we use an alternative routine which is passed a floating-point approximation to c^{-1} as a parameter, thus replacing an (expensive) division by a (less expensive) multiply. Moreover, one of the multiplicands, say b, is often fixed for many such operations (this is the case in the inner loop of the FFT), and in this case we use an alternative routine which is passed a floating-point approximation to bc^{-1} as a parameter, thus eliminating one multiplication entirely, so that one multiply mod c costs just one floating-point multiply and two integer multiplies.

For the alternative version of LIP, where $R = 2^{26}$, we use the routine in Figure 5. As above, when c remains fixed, we pre-compute c^{-1} and pass it as a parameter; in this case, the cost of a multiply mod c is three floating-point multiplies.

Figure 5: Modular multiply with 26-bit radix

```
inline long MulMod(long a, long b, long c)
{
   double prod = double(a)*double(b);
   long q = long(prod/double(c));
   long r = long(prod - double(q)*double(c));
   if (r < 0) r = r + c;
   else if (r >= c) r = r - c;
   return r;
}
```

Figure 6: Double-precision integer remainder

```
inline long rem21(long hi, long lo, long c)
{
   long q = long((double(hi)*double(1 << 30) + double(lo))/double(c));
   long r = (hi << 30) + lo - q*c;
   if (r < 0) r = r + c;
   else if (r >= c) r = r - c;
   return r;
}
```

8.3 Multi-precision/single-precision division

Another critical operation in our polynomial arithmetic algorithms is the reduction of a multi-precision number modulo a single-precision number.

The default 30-bit version of LIP does this as follows. The routine `rem21` in Figure 6 shows how a double-precision integer (hi, lo) is reduced modulo a single-precision value c, assuming $hi < c$. By applying this routine repeatedly, we can easily reduce an arbitrary multi-precision number modulo c. For efficiency, we pre-compute c^{-1} and use a version of the `rem21` routine that takes this as a parameter, so as to avoid repeated divisions.

In the alternative version of LIP, where $R = 2^{26}$, for each c that will be used in such an operation (i.e., all of the FFT-primes $q_1, \ldots, q_\ell$), we pre-compute a table of powers

$$r_0 = 1,\ r_1 = (R \bmod c),\ r_2 = (R^2 \bmod c),\ \ldots,$$

and then to reduce a number $\sum_i d_i R^i$ modulo c, we compute the sum $S = \sum_i d_i r_i$ as a triple-precision integer using the same extraction and pipelining techniques as described in §8.1. Then we reduce the sum $S \bmod c$ with two applications of a 26-bit version of the `rem21` routine described above.

8.4 Chinese Remaindering

The final critical sub-task in our polynomial arithmetic over $\mathbf{F}_p$ is the application of the Chinese Remainder Theorem. In our situation, we have ℓ single-precision primes $q_1, \ldots, q_\ell$, and we are given integers $a_1, \ldots, a_\ell$ such that $0 \le a_i < q_i$ for $1 \le i \le \ell$. The task is to compute $(a \bmod p)$, where a is the unique integer in the interval $(-P/2, P/2)$, where $P = \prod q_i$, satisfying the congruences

$$a \equiv a_i \pmod{q_i} \quad (1 \le i \le \ell).$$

As is standard, for $1 \le i \le \ell$, we pre-compute z_i as the multiplicative inverse of $(P/q_i) \bmod q_i$, and we pre-compute the CRT-coefficients $m_i = z_i(P/q_i)$.

Then to compute a, we could compute $S = \sum_i a_i m_i$, and then reduce S modulo P, computing the residue of least absolute value, obtaining a. Once we have a, we could compute $(a \bmod p)$.

The numbers m_i $(1 \le i \le \ell)$ and P are a little more than $2 \log_2 p$ bits in length, and we avoid working with such large numbers in our implementation with the following technique, which in practice takes about half the time as the above method.

First of all, we assume that the solution a to the system of congruences satisfies $a \in (-P/4, P/4)$. Enforcing this assumption requires us to increase the required lower bound on the product P by 1 bit when selecting the FFT-primes. This assumption implies that the distance between S/P and its nearest integer is less than $1/4$.

Second of all, note that

$$\begin{aligned} a &= S - P\lfloor 0.5 + S/P \rfloor \\ &= \sum_i a_i m_i - P\left\lfloor 0.5 + \sum_i a_i (z_i/q_i) \right\rfloor. \end{aligned}$$

We pre-compute $(-P \bmod p)$, and for $1 \le i \le \ell$, we also pre-compute $(m_i \bmod p)$ and a floating-point approximation to z_i/q_i. To compute $(a \bmod p)$, we then compute

$$t = \sum_i a_i (m_i \bmod p) + (-P \bmod p)\left\lfloor 0.5 + \sum_i a_i (z_i/q_i) \right\rfloor,$$

and then reduce $t \bmod p$.

To compute t using the above formula, the first sum requires ℓ multiplies of single-precision and $\log_2 p$ bit numbers. The second sum is computed using ℓ floating-point multiplies and additions. Because of our assumption, the value of this sum is close to an integer, and so when we round to the nearest integer, the result will be correct, despite the round-off errors inherent in floating-point arithmetic.

Since t has just a few more than $\log_2 p$ bits, reducing $t \bmod p$ is not very costly.

9 Experimental Results

In this section, we report the results of our experiments with our software. *All of the timing results, except where explicitly noted, were obtained on a Sun SPARC-station ELC.*

9.1 Multi-precision Arithmetic

We compared our multi-precision integer software (see §8) on several machines against hand-coded assembly. The machines we used were a Sun SPARC-station ELC, a Sun SPARC-station 10, and a Siemens-Nixdorf RW420, which has a MIPS R4000 processor. All of these machines have 32-bit

Table 3: Multi-precision Integer Multiplication

	libI	LIP-26	LIP-26*	LIP-30
SPARC-ELC	42	39	52	112
SPARC-10	9	14	21	28
MIPS	6	12	15	26

RISC processors with fast floating-point units that run in parallel with the integer unit. The main difference between the SPARC-ELC and SPARC-10, besides clock speed, is that the SPARC-10 has an integer multiply instruction (which computes the 64 bit product), whereas integer multiplication on the SPARC-ELC must be done in software. The MIPS processor also has an integer multiply instruction.

For the assembly-code, we used the software library `libI` (version 2.1), written by Ralf Dentzer [7]. This library is written in `C`, but contains highly optimized assembly-language code for the equivalent of our `AddMul` routine. For multi-precision integer multiplication, on these machines, `libI` is as fast or faster than just about any other available software.

As our benchmark, we simply multiply two random 100 digit (332 bit) numbers together, repeating this 200,000 times. The timing results for our benchmark are shown in Table 3. Time is indicated in seconds. The column labeled *libI* shows the time for the libI assembly-code program. The column labeled *LIP-26* shows the running time for our alternative, 26-bit version of LIP. The column labeled *LIP-26** shows the running time for the 26-bit version, but without the software pipeline. The column labeled *LIP-30* shows the running time for the default, 30-bit version of LIP.

The running times show that on these machines, the explicit software pipeline in LIP-26 indeed has a significant impact on performance. They also show that the 26-bit alternative version of LIP is quite competitive with hand-coded assembly on these machines; indeed, LIP is even a little faster than libI on the SPARC-ELC.

All other timing results reported in this paper are based on the LIP-26 arithmetic.

9.2 Arithmetic in $\mathbf{F}_p[x]$

In this subsection, we report timing results for our algorithms for polynomial arithmetic in $\mathbf{F}_p[x]$ (see §7). These results are reported in Tables 4, 5, and 6. In each of these tables, for various n, operations were performed on polynomials over $\mathbf{F}_p$ of degree near n, the difference between the tables being the choice of p. In Table 4, p is a random 100 digit (332 bit) prime, in Table 5, p is a 200 digit (664 bit) prime, and in Table 6, p varies with n, so that p is a random n bit prime.

Running times are reported in seconds. Operations that required only a very small amount of time were iterated several times, taking an average. For the various n, the following operations were performed using a random monic polynomial $f \in \mathbf{F}_p[x]$ of degree n, two random polynomials $g, h \in \mathbf{F}_p[x]$ of degree $n-1$, and one random polynomial $G \in \mathbf{F}_p[x]$ of degree $2n-1$.

plain mul: compute $g \cdot h$ with classical algorithm;

mul: compute $g \cdot h$ with FFT-based algorithm; also a timing breakdown is shown for the sub-tasks *RED* (reducing coefficients modulo the FFT-primes), *CRT* (applying the Chinese Remainder Theorem to the coefficients), and *FFT* (performing the FFT);

Table 4: Arithmetic in $\mathbf{F}_p[x]$, $\log_2 p = 332$

n	64	128	256	512	1,024	2,048
plain mul	0.99	3.86	15.29	61.19	246.63	985.08
mul	0.28	0.58	1.23	2.54	5.32	11.27
RED	0.07	0.14	0.27	0.55	1.08	2.15
CRT	0.08	0.16	0.33	0.65	1.30	2.60
FFT	0.13	0.27	0.59	1.28	2.79	6.29
div	0.96	1.61	3.44	7.19	15.16	31.57
inv	0.39	0.78	1.65	3.42	7.20	14.98
mod f	0.28	0.59	1.25	2.59	6.24	11.22
mul mod f	0.52	1.10	2.32	4.83	11.02	20.93
mul by h mod f	0.29	0.60	1.36	2.63	5.90	11.50
transposed mul	0.27	0.58	1.20	2.53	5.25	10.92
sqr mod f	0.45	0.94	1.97	4.11	9.45	17.80
plain GCD	3.67	10.70	34.65	121.79	457.38	1,785.34
fast GCD	3.66	10.43	26.41	64.79	159.50	360.91

div: divide G by f, with remainder (no pre-conditioning);

inv: compute $g^{-1} \bmod x^{-n}$;

mod f: compute $G \bmod f$, pre-conditioned on f;

mul mod f: compute $g \cdot h \bmod f$, pre-conditioned on f;

mul by h mod f: compute $g \cdot h \bmod f$, pre-conditioned on both f and h;

transposed mul: compute transposition of *mul by h mod f*;

sqr mod f: compute $g^2 \bmod f$, pre-conditioned on f;

plain GCD: compute $\gcd(f, g)$ with Euclid's algorithm;

fast GCD: compute $\gcd(f, g)$ with the asymptotically fast "Half-GCD" algorithm.

Several conclusions can be inferred from this data. First of all, our FFT-based multiplication algorithm is significantly faster than classical multiplication, already at degree 64 (the cross-over is actually around 30). Second, the data verifies our analytical estimate that the operations *mul*, *mul by h mod f*, and *transposed mul* take very nearly the same amount of time. Third, the data shows that computing polynomial inverses takes less than 1.5 times the time of a polynomial multiplication, and a modular squaring takes a little more time, but still less than twice the time of a polynomial multiplication. This is in general agreement with our analytical estimates. Fourth, the data indicates that the asymptotically fast GCD algorithm does indeed run faster than Euclid's algorithm, but only for reasonably large n (≥ 256). We remark that in our implementation, for $n \leq 128$, the *plain GCD* and *fast GCD* are really the same algorithm; that is why the running times are nearly identical.

Table 5: Arithmetic in $\mathbf{F}_p[x]$, $\log_2 p = 664$

n	64	128	256	512	1,024	2,048
plain mul	3.56	14.02	55.95	222.96	890.60	3,559.77
mul	0.69	1.44	2.98	6.16	12.71	26.73
RED	0.20	0.40	0.80	1.63	3.23	6.43
CRT	0.24	0.49	0.98	1.94	3.89	7.85
FFT	0.23	0.52	1.13	2.46	5.33	11.93
div	3.38	4.02	8.34	17.28	35.91	74.83
inv	1.08	1.99	4.06	8.23	17.00	35.30
mod f	0.70	1.45	3.30	6.22	12.90	26.77
mul mod f	1.28	2.65	5.52	11.41	23.69	49.31
mul by h mod f	0.72	1.51	3.08	6.29	13.01	27.10
transposed mul	0.72	1.47	3.04	6.28	13.18	27.81
sqr mod f	1.09	2.30	4.73	9.78	20.32	42.14
plain GCD	10.26	24.57	100.37	365.97	1,445.80	5,778.22
fast GCD	11.36	20.43	55.58	139.12	336.84	803.28

Table 6: Arithmetic in $\mathbf{F}_p[x]$, $\log_2 p = n$

n	64	128	256	512	1,024
plain mul	0.11	0.85	9.36	128.79	1,802.37
mul	0.05	0.20	0.88	4.61	24.20
RED	0.00	0.02	0.15	1.29	7.24
CRT	0.02	0.05	0.24	1.30	8.43
FFT	0.02	0.11	0.46	1.91	8.14
div	0.11	0.53	2.43	12.97	67.87
inv	0.05	0.24	1.14	6.04	31.98
mod f	0.05	0.21	0.90	4.67	24.50
mul mod f	0.08	0.38	1.65	8.67	44.49
mul by h mod f	0.05	0.21	0.91	4.89	24.63
transposed mul	0.05	0.21	0.90	4.51	24.54
sqr mod f	0.07	0.33	1.42	7.36	38.09
plain GCD	0.29	1.85	18.32	228.45	3,067.13
fast GCD	0.29	2.21	14.19	99.23	666.64

Table 7: Modular composition and related problems, $\log_2 p = n$

n	64	128	256	512	1,024
compose	0.90	5.62	38.42	333.41	3,417.08
min poly	2.37	13.32	84.83	680.48	6,952.24
trace n/2	6.84	49.88	423.15	4,360.31	56,007.50
trace n/2-1	7.40	56.98	500.17	5,380.73	72,677.70

9.3 Modular Composition and Related Problems

Next we present timing results for modular composition (§2.1), minimum-polynomial computation (§4), and trace-map computation (§3.1). These are given in Table 7. Time is reported in seconds. For various values of n, a random n-bit prime p was chosen, and the following computations were performed using a random monic polynomial $f \in \mathbf{F}_p[x]$ of degree n, and two random polynomials $g, h \in \mathbf{F}_p[x]$ of degree n.

compose: compute $g(h) \bmod f$;

min poly: compute the minimum polynomial of h modulo f;

trace n/2: compute the trace map $T_{n/2}(h) \bmod f$, given $x^p \bmod f$;

trace n/2-1: compute the trace map $T_{n/2-1}(h) \bmod f$, given $x^p \bmod f$.

In all cases, the power-table size was chosen so as to minimize the running time, except for the case $n = 1024$, where the size was limited to 25 polynomials to keep the space requirement small.

The reason for computing T_d for both $d = n/2$ and $d = n/2 - 1$ is that the running time for computing T_d is highly sensitive to the number of 1-bits in d, and hence for n a power of two, the two choices of d reflect the best-case and worst-case inputs, respectively.

One interesting thing to note is the performance of our new minimum-polynomial algorithm in comparison with the technique of simply computing successive powers and finding a linear relation using Gaussian elimination. We have not implemented this latter method, but from our other timing data, we can reasonably estimate the corresponding running times for this algorithm as roughly

$$6, \quad 64, \quad 1,100, \quad 24,800, \quad 640,000$$

seconds for the values of n in our table. Thus, our new algorithm is significantly faster than this method, and in fact uses much less space as well.

One might also compare our new minimum-polynomial algorithm with one which is identical to ours, except that the power projection step is implemented naively. For example, in the case $n = 1,024$, we estimate that such an algorithm would use about $91,000$ seconds just to compute successive powers mod f.

9.4 Factoring Polynomials

Finally, we present our timing data for polynomial factorization.

Table 8: Factorization pattern of F_n mod P_n

n	degrees of factors
64	1, 4, 6, 7, 17, 29
128	2, 2, 124
256	1, 1, 1, 1, 2, 5, 102, 143
512	3, 3, 9, 497
1024	1, 1, 2, 5, 42, 42, 44, 49, 77, 120, 198, 443
2048	2, 28, 69, 884, 1065

Joachim von zur Gathen [20] has proposed a set of benchmarks by which to judge polynomial factorization algorithms. For every n, von zur Gathen's nth benchmark is to factor $x^n + x + 1$ modulo the first prime $\geq \lfloor 2^n \pi \rfloor$. A drawback with this proposal is that one might feel encouraged to exploit the sparseness of the polynomial to speed things up.

We propose a different set of benchmarks that is in the same spirit as von zur Gathen's, but which corrects this drawback. Define the sequence of integers $a_0, a_1, a_2, \ldots,$ by the recurrence $a_0 = 1$, and $a_{i+1} = a_i^2 + 1$ for $i \geq 0$. For $n \geq 2$, define the polynomial

$$F_n = \sum_{i=0}^{n} a_i x^{n-i},$$

which is a monic polynomial of degree n. Also, define P_n to be the first prime $\geq \lfloor 2^{n-2}\pi \rfloor$, which is an n-bit prime. Our nth benchmark is to factor F_n modulo P_n.

Like von zur Gathen's proposed test problems, ours are easy to describe, easy to generate, and yet appear to behave as "generic" polynomials, as far as factoring is concerned. Our benchmarks also are a bit more aesthetically appealing, as P_n has exactly n bits, rather than $n + 2$.

For powers of two between 64 and 2048, we have factored F_n mod P_n. All of these polynomials are square-free. Table 8 shows the degrees of the irreducible factors of these polynomials.

Table 9 shows the time and space requirements for our implementation of the new algorithm and of the null-space method. Time is recorded in seconds, space in megabytes of main memory.

For the new algorithm, we report the total running time, as well as the running time for various sub-tasks: *square-free* indicates the time used in square-free decomposition stage (just a GCD); x^p indicates the time spent in step 1 of the new distinct-degree factorizer (see §2); *baby/giant steps* indicates the time spent in Steps 2 and 3 of that algorithm; *refine* indicates the time spent in Steps 4 and 5; *EDF* indicates the time spent performing equal-degree factorizations; *space* indicates the main memory requirement.

For the null-space method (see §5), we report the total running time, as well as the time for various sub-tasks: *square-free* indicates the time used in square-free decomposition stage; x^p indicates the time spent in step 1 of our implementation of the null-space method; *build matrix* indicates the time spent in Step 2 of that algorithm; *Gauss* indicates the time spent in step 3; *subst/extract* indicates the time spent in Steps 4 and 5; *space* indicates the main memory requirement.

For the new algorithm, in the baby step/giant step computation, for the modular compositions we used the default power-table size of $2\lfloor\sqrt{n}\rfloor$, except for $n = 1024$, where we overrode the default and used a power-table size of only 25 so as to keep the space requirement reasonably small.

Table 9: Factoring F_n modulo P_n (SPARC-ELC)

n	64	128	256	512	1,024
New Algorithm	12.7	112.4	1,089.7	12,308.4	183,466.0
square-free	0.3	2.1	14.7	93.8	677.5
x^p	4.5	39.4	368.1	3,595.5	41,465.0
baby/giant steps	5.1	40.0	438.8	6,002.6	125,931.3
refine	2.5	29.9	261.8	2,602.0	14,685.5
EDF	0.0	1.0	4.8	4.9	613.73
space	0.6	0.8	1.5	4.4	11.0
Null-Space Method	13.0	111.1	1,530.8	28,732.6	
square-free	0.3	2.2	14.1	94.2	
x^p	4.5	39.6	363.8	3,604.4	
build matrix	3.3	26.1	236.0	2,305.4	
Gauss	2.7	38.0	850.7	22,528.7	
subst/extract	2.0	4.9	64.8	192.3	
space	0.7	1.6	6.1	47.5	

We point out that Table 9 does not show the amount of disk space used by the new algorithm, which is generally bit more than the amount of main memory used. In the case $n = 1024$, for example, the new algorithm used about 15 MB disk space (this could easily be cut in half). Disk access time was not significant for the new algorithm, as the number of disk reads and writes is quite minimal. The null-space method does not explicitly use any external storage.

As the data shows, for $n \leq 128$ the running times of the two algorithms are very similar. For $n \geq 256$, the new algorithm starts to exhibit a significant advantage over the null-space method in terms of running time. However, more significant is the space advantage. Indeed, we were unable to run the null-space method on the $n = 1024$ benchmark on any of our machines due to space limitations: it would require over 300 MB to store the matrix in our implementation. One can, however, quite reasonably estimate the time to diagonalize the matrix as $n/3$ times the time to multiply two polynomials of degree $n-1$ using the classical algorithm. Using the data in Table 6, this leads to an estimate of roughly 615,000 seconds to diagonalize the matrix in the $n = 1024$ benchmark.

The data for the null-space method also shows the advantage of the minimum-polynomial extraction method over the simple extraction method (see §3.2). The running time for the simple extraction method would have been larger than the running time for the x^p computation, which is many times larger than the extraction time shown in the table.

Some time after our original experiments were carried out, we obtained access to a Sun SPARC-station 10, which is between 2 and 3 times faster than the ELC, and also has more memory than the ELC. It was on the SPARC-10 that we factored the $n = 2048$ benchmark. The time and space results are reported in Table 10. For the power-table size parameter in the new algorithm, we overrode the default only in the $n = 2048$ case, setting it to 64.

Another interesting family of factorization test cases are polynomials of the form $f = f_1 f_2$, where f_1 and f_2 are distinct irreducible polynomials of the same degree, $n/2$. This class of problems is interesting for two reasons. First, it exhibits the worst-case behavior of the new algorithm, as

Table 10: Factoring F_n modulo P_n (SPARC-10)

n	64	128	256	512	1,024	2,048
New Algorithm	5.3	45.9	412.2	4,634.9	60,298.0	979,740.0
square-free	0.1	0.8	5.3	34.7	262.3	1,972.6
x^p	1.8	16.1	139.8	1,388.7	16,877.3	209,228.0
baby/giant steps	2.1	16.3	162.1	2,203.1	36,836.6	649,532.0
refine	1.1	12.3	102.5	1,002.9	6,047.5	118,894.2
EDF	0.0	0.2	1.8	1.8	235.6	0.0
space	0.6	0.8	1.5	4.4	18.3	68.0
Null-Space Method	5.2	44.8	559.1	9,942.4		
square-free	0.1	0.9	5.3	34.3		
x^p	1.8	16.2	140.2	1,396.6		
build matrix	1.3	10.8	90.4	886.6		
Gauss	1.1	14.8	298.6	7,552.8		
subst/extract	0.7	1.9	24.1	69.0		
space	0.7	1.6	6.1	47.5		

we have to perform a complete distinct-degree factorization, followed by a very large equal-degree factorization problem. Second, in some situations we know that f is of this special form, or at least of the form $f = f_1 \dots f_r$, where all the f_i's have the same degree, and in this case we can by-pass the distinct-degree factorization step altogether, and (after computing x^p) proceed directly to the equal-degree factorization stage.

When the equal-degree factorizer is applied to $f = f_1 f_2$ as above, almost all of the time is spent computing the trace map. The data in Table 7 tells us how long that would take for polynomials whose sizes are similar to those considered in that table.

We also made some empirical tests. Table 11 shows timing results for polynomials of the form $f = f_1 f_2$, where $n = \deg(f)$. The primes were each 201-bits in length, and the polynomials f_1 and f_2 were generated randomly. For comparison, we also ran the null-space method—whose running time is fairly insensitive to the factorization pattern—on these inputs. The format of this table is the same as that of Table 9, except that we have expanded the EDF row to show separately the time to compute the trace map and the time to perform factor extraction.

Since the new algorithm exhibits extreme worst-case behavior on these inputs, the cross-over point relative to the null-space method is higher than is typical for random inputs. But notice that the new algorithm still uses significantly less space, which allowed us to factor the $n = 1004$ case with our algorithm, whereas the null-space method on this input would have exceeded the space limitations on our available machines. Also, if we knew f had this special form, we could subtract off the *baby/giant steps* time and the *refine* time from the running time of the new algorithm. If we do this, this gives us running times of

$$67.7, \quad 165.7, \quad 578.1, \quad 1,760.8, \quad 5,928.3$$

seconds for the values of n in our table. So, for example, the new algorithm would be 3 times faster than the null-space method at $n = 500$.

Table 11: Factoring $f = f_1 \cdot f_2$ modulo a 201-bit prime

n	56	104	252	500	1,004
New Algorithm	92.4	259.7	1,109.5	4,040.5	16,561.3
square-free	1.0	2.8	10.0	25.7	65.0
x^p	45.0	90.1	205.5	428.8	905.1
baby/giant steps	13.4	51.5	309.6	1,426.1	7,037.5
refine	11.3	42.5	221.8	853.6	3,595.5
trace map	18.8	65.7	344.2	1,267.6	4,871.7
extract	2.4	5.7	15.5	32.7	73.8
space	1.0	1.0	1.0	2.0	5.0
Null-Space Method	63.6	172.5	971.9	5,603.0	
square-free	0.8	2.4	10.2	26.5	
x^p	45.0	91.9	208.9	434.2	
build matrix	8.1	30.4	170.0	717.0	
Gauss	6.8	41.0	563.1	4,375.8	
subst/extract	2.6	6.4	18.5	47.1	
space	1.0	2.0	6.0	21.0	

10 Conclusion

There are two final topics upon which we briefly comment.

The first topic concerns application of the baby step/giant step technique to the black-box variant of the null-space method. We refer the reader to [11] for the details of how this is done. From a computational complexity point of view, the main difference between the null-space approach and the degree-separation approach is the following: in a straightforward implementation, the null-space approach would require between about 2 and 3 times as many modular compositions (or power projections) as the degree-separation approach; offsetting this is the fact that the work done in the "refine" steps of the distinct-degree factorizer (steps 4 and 5) does not need to be done in the null-space approach. Our experience indicates that (with our current knowledge) the degree-separation approach is slightly more efficient than the null-space approach, in terms of both time and space.

The second topic concerns the use of classical, quadratic-time algorithms for polynomial arithmetic. While we believe our experimental results very clearly demonstrate the practical superiority of FFT arithmetic over classical arithmetic, it unfortunately seems likely that many general-purpose computer algebra systems will continue to use only classical arithmetic for some time to come. If classical algorithms are used, the new baby step/giant step technique, applied to the degree-separation method, yields an algorithm that uses

$$O(n^3 + n^2 \log p)$$

scalar operations, and space for $O(n^{1.5})$ scalars. This itself may be of some value, as it matches the running time for the classical implementation of the null-space method, while reducing the space complexity significantly. The n^3 term cannot be reduced by trading time for space, as the "refine" steps of the distinct-degree factorizer must perform $\Theta(n)$ polynomial multiplications. Indeed, under a "classical" efficiency metric, the refine steps will dominate the running time of the algorithm.

Applying the baby step/giant step technique to the black-box variant of the null-space method allows for an interesting time/space trade-off. This arises because of the lack of the "refine" steps, as mentioned above. Using classical algorithms, this method uses

$$O(tn^2 + n^{3.5}/t + n^2 \log p)$$

scalar operations, and space for $O(n^{1.5} + tn)$ scalars, where $1 \leq t \leq n$ is a parameter. Optimizing the time, this yields

$$O(n^{2.75} + n^2 \log p)$$

scalar operations, and space for $O(n^{1.75})$ scalars, which beats the classical implementation of the null-space method in terms of both time and space.

We have not implemented the black-box variant of the null-space method, nor have we implemented any of the above factoring methods based on classical arithmetic. It may be useful to obtain some practical experience with these algorithms.

In conclusion, we believe our empirical results justify our claim that the new algorithm and its implementation allow much larger polynomials to be factored than was previously possible. We also believe that many of the techniques developed here can be applied to problems other than factoring polynomials over finite fields, and will be useful to other implementors of computer algebra software. In particular, we hope that our work will encourage implementors of general-purpose computer algebra systems to implement FFT-based polynomial arithmetic.

Acknowledgements

The author would like to thank Erich Kaltofen for sharing many of his ideas on this topic with me, Mark Giesbrecht for his valuable comments on an earlier draft of this paper. This research was supported by an Alexander von Humboldt Fellowship while the author was visiting the Universität des Saarlandes, Germany.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] E. R. Berlekamp. Factoring polynomials over large finite fields. *Math. Comp.*, 24(111):713–735, 1970.

[3] W. Bosma, J. Cannon, C. Playoust, and A. Steel. Solving problems with MAGMA. Preprint, 1994.

[4] R. P. Brent and H. T. Kung. Fast algorithms for manipulating formal power series. *J. Assoc. Comput. Mach.*, 25:581–595, 1978.

[5] D. G. Cantor and H. Zassenhaus. A new algorithm for factoring polynomials over finite fields. *Math. Comp.*, 36(154):587–592, 1981.

[6] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[7] R. Dentzer. The libI software library. Available via anonymous FTP from `ftp.iwr.uni-heidelberg.de` in `pub/IntArith`, 1994.

[8] J. L. Dornstetter. On the equivalence between Berlekamp's and Euclid's algorithms. *IEEE Trans. Inf. Theory*, IT-33:428–431, 1987.

[9] E. Kaltofen and A. Lobo. Factoring high-degree polynomials by the black box Berlekamp algorithm. In *Proc. Internat. Symp. Symbolic Alebraic Comput.*, 1994.

[10] E. Kaltofen and V. Pan. Processor efficient parallel solution of linear systems over an abstract field. In *Proc. 3rd Ann. ACM Symp. Parallel Algor. Architecture*, pages 180–191, 1991.

[11] E. Kaltofen and V. Shoup. Subquadratic-time factoring of polynomials over finite fields. In *27th Annual ACM Symposium on Theory of Computing*, pages 398–406, 1995.

[12] M. Kaminski, D. G. Kirkpatrick, and N. H. Bshouty. Addition requirements for matrix and transposed matrix products. *Journal of Algorithms*, 9:354–364, 1988.

[13] F. Lehmann, M. Maurer, V. Müller, and V. Shoup. Counting the number of points on elliptic curves over finite fields of characteristic greater than three. In *First Algorithmic Number Theory Symposium*, 1994.

[14] A. K. Lenstra. Documentation of LIP. Available via anonymous FTP from `flash.bellcore.com` in `pub/lenstra`, 1994.

[15] R. Lidl and H. Niederreiter. *Finite Fields*. Addison-Wesley, 1983.

[16] M. Monagan. Von zur Gathen's factorization challenge. *SIGSAM Bulletin*, 27(2):13–18, 1993.

[17] P. Montgomery. *An FFT Extension of the Elliptic Curve Method of Factorization*. PhD thesis, Univ. of California–Los Angeles, 1992.

[18] F. Morain. Implantation de l'algorithme de Schoof-Elkies-Atkin. Preprint, 1994.

[19] V. Shoup. Fast construction of irreducible polynomials over finite fields. *J. Symbolic Comp.*, 17:371–391, 1994.

[20] J. von zur Gathen. A polynomial factorization challenge. *SIGSAM Bulletin*, 26(2):22–24, 1992.

[21] J. von zur Gathen and V. Shoup. Computing Frobenius maps and factoring polynomials. *Computational Complexity*, 2:187–224, 1992.

[22] D. Wiedemann. Solving sparse linear systems over finite fields. *IEEE Trans. Inf. Theory*, IT-32:54–62, 1986.

[23] D. Y. Y. Yun. On square-free decomposition algorithms. In *Proc. ACM Symp. Symbolic and Algebraic Comp.*, pages 26–35, 1976.