A high-performance algorithm for calculating cyclotomic polynomials.

Andrew Arnold
Department of Mathematics
Simon Fraser University
Burnaby, B.C. Canada.
ada26@sfu.ca.

Michael Monagan
Department of Mathematics
Simon Fraser University
Burnaby, B.C. Canada.
mmonagan@cecm.sfu.ca.

ABSTRACT

The n_{th} cyclotomic polynomial, $\Phi_n(z)$, is the monic polynomial whose $\phi(n)$ distinct roots are the n_{th} primitive roots of unity. $\Phi_n(z)$ can be computed efficiently as a quotient of terms of the form $(1 - z^d)$ by way of a method the authors call the Sparse Power Series algorithm. We improve on this algorithm in three steps, ultimately deriving a fast, recursive algorithm to calculate $\Phi_n(z)$. The new algorithm, which we have implemented in C, allows us to compute $\Phi_n(z)$ for $n > 10^9$ in less than one minute.

Categories and Subject Descriptors:
G.0 [Mathematics of Computing]: General
General Terms: Algorithms
Keywords: Cyclotomic Polynomials

1. INTRODUCTION

The n_{th} cyclotomic polynomial, $\Phi_n(z)$, is the minimal polynomial over $\mathbb{Q}$ of the n_{th} primitive roots of unity.

$$\Phi_n(z) = \prod_{\substack{j=1 \\ \gcd(j,n)=1}}^{n} \left(z - e^{\frac{2\pi i}{n} j}\right). \tag{1.1}$$

We let the *index* of $\Phi_n(z)$ be n and the *order* of $\Phi_n(z)$ be the number of distinct odd prime divisors of n. The n_{th} inverse cyclotomic polynomial, $\Psi_n(z)$, is the monic polynomial whose roots are the n_{th} non-primitve roots of unity.

$$\Psi_n(z) = \prod_{\substack{j=1 \\ \gcd(j,n)>1}}^{n} \left(z - e^{\frac{2\pi i}{n} j}\right) = \frac{z^n - 1}{\Phi_n(z)}. \tag{1.2}$$

We denote by $A(n)$ the *height* of $\Phi_n(z)$, that is, the largest coefficient in magnitude of $\Phi_n(z)$. It is well known that for $n < 105$, $A(n) = 1$ but for $n = 105$, $A(n) = 2$. The smallest n with $A(n) > 2$ is $n = 385$ where $A(n) = 3$. Although the heights appear to grow very slowly, Paul Erdős proved in [2] that $A(n)$ is not bounded above by any polynomial in n, that is, for any constant $c > 0$, there exists n such that $A(n) > n^c$. Maier showed that the set of n for which $A(n) > n^c$ has positive lower density. A natural question to ask is, what is the first n for which $A(n) > n$?

In earlier work [1], we developed two asymptotically fast algorithms to compute $\Phi_n(z)$. The first algorithm, which we call the FFT algorithm, uses the Fast Fourier Transform to perform a sequence of polynomial exact divisions in $\mathbb{Z}[z]$ modulo a prime q. Using this algorithm we found the smallest n such that $A(n) > n$, namely for $n = 1,181,895$, the height $A(n) = 14,102,773$. Here $n = 3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29$. To find $\Phi_n(z)$ with larger height, we tried simply multiplying this n by additional primes. In this way we found an n with $A(n) > n^2$ and several $n > 10^9$ with $A(n) > n^4$, the latter requiring the use of a supercomputer with a lot of RAM.

The second algorithm, which we call the Sparse Power Series (SPS) algorithm, does a sequence of sparse series multiplications and divisions in $O(2^k\phi(n))$ integer arithmetic operations. Although not asymptotically faster than the FFT algorithm, it turns out that because the SPS algorithm only needs integer additions and subtractions, it is considerably faster (more than 20 times - see section 4) than the FFT algorithm. Using the SPS algorithm we found the smallest n with $A(n) > n^2$, $A(n) > n^3$ and $A(n) > n^4$, namely $n = 43,730,115$, $n = 416,690,995$, and $1,880,394,945$, respectively, as well as other new results. One of the difficulties when $n > 10^9$ is space. For such n, even storing $\Phi_n(z)$ requires many gigabytes of memory. The SPS algorithm has a substantial space advantage over the FFT algorithm. It has now been implemented in the Sage and Maple 13 computer algebra systems.

In this paper we present a fast recursive algorithm to calculate $\Phi_n(z)$ and $\Psi_n(z)$. It improves on the sparse power series (**SPS**) algorithm by approximately another factor of 10 (see section 4). To give one specific benchmark; Yoichi in [4, 5] found $A(n)$ for n the product of the first 7 odd primes but was unable to determine $A(n)$ for n the product of the first 8 primes. We used the FFT algorithm to find $A(n)$ for n the product of the first 9 odd primes in approx. 12 hours. The SPS algorithm takes 7 minutes and our new algorithm takes 50 seconds. A challenge problem given to us by Noe [6] is to compute $\Phi_n(z)$ for $n = 99,660,932,085$ which we expect will have a huge height. The main difficulty now is space; for the mentioned unsolved problem, the set of coefficients of $\Phi_n(z)$, stored as 320-bit integers (which we estimate will be sufficient) requires over 750 GB of space.

Our paper is organized as follows. Section 2 presents identities involving $z^n - 1$, $\Phi_n(z)$ and $\Psi_n(z)$ used in the algorithms, and the basic algorithm used for the FFT approach.

Section 3 details the Sparse Power Series algorithm for computing $\Phi_n(z)$ and introduces a similar algorithm for computing $\Psi_n(z)$, then develops improvements in three steps. The third step makes the algorithm recursive. Section 4 presents some timings comparing the FFT algorithm, the original SPS algorithm, and the three improvements.

2. USEFUL IDENTITIES OF CYCLOTOMIC POLYNOMIALS

Before describing the algorithms, we establish some basic identities of cyclotomic polynomials. First, as the roots of $\Phi_n(z)$ and $\Psi_n(z)$ consist of all n_{th} roots of unity, we have

$$\Phi_n(z)\Psi_n(z) = \prod_{j=0}^{n-1}(z - e^{\frac{2\pi j}{n}i}) = z^n - 1. \qquad (2.1)$$

Every n_{th} root of unity is a d_{th} primitive root of unity for some unique $d|n$. Conversely, if $d|n$, every d_{th} primitive root of unity is trivially an n_{th} root of unity. As such,

$$\prod_{d|n}\Phi_d(z) = z^n - 1. \qquad (2.2)$$

Applying the Möbius inversion formula to (2.2), we have

$$\Phi_n(z) = \prod_{d|n}(z^d - 1)^{\mu(\frac{n}{d})}, \qquad (2.3)$$

where μ is the Möbius function. From (2.1) and (2.3) we obtain a similar identity for $\Psi_n(z)$.

$$\Psi_n(z) = \prod_{d|n, d<n}(z^d - 1)^{-\mu(\frac{n}{d})}, \qquad (2.4)$$

and from (2.1) and (2.2), we have that

$$\Psi_n(z) = \prod_{d|n, d<n}\Phi_d(z). \qquad (2.5)$$

Thus $\Psi_n(z)$ is a product cyclotomic polynomials.

Given $\Phi_1(z) = z - 1$ and $\Psi_1(z) = 1$, we can compute all cyclotomic polynomials using the following lemmas.

Lemma 1. *If p, q primes such that $p \nmid n$ and $q|n$ then*

$$\Phi_{np}(z) = \frac{\Phi_n(z^p)}{\Phi_n(z)}, \qquad (2.6a)$$

$$\Phi_{nq}(z) = \Phi_n(z^q), \qquad (2.6b)$$

$$\Psi_{np}(z) = \Psi_n(z^p)\Phi_n(z), \text{ and} \qquad (2.6c)$$

$$\Psi_{nq}(z) = \Psi_n(z^q). \qquad (2.6d)$$

Lemma 2. *If $n > 1$ is odd then*

$$\Phi_{2n}(z) = \Phi_n(-z) \text{ and} \qquad (2.7a)$$

$$\Psi_{2n}(z) = -\Psi_n(-z)(z^n + 1). \qquad (2.7b)$$

Lemmas 1 and 2 are well-known. One can prove these identities by equating roots of both sides of the respective equations. Given $\Phi_n(z)\Psi_n(z) = z^n - 1$, the identities for $\Psi_n(z)$ (2.6c), (2.6d) and (2.7b) can be easily derived from (2.6a), (2.6b) and (2.7a), their respective analogs for $\Phi_n(z)$.

These lemmas give us a means to calculate $\Phi_n(z)$. For example, for $n = 150 = 2 \cdot 3 \cdot 5^2$ we have

$$\begin{aligned}
\Phi_3(z) &= \tfrac{\Phi_1(z^3)}{\Phi_1(z)} = \tfrac{z^3-1}{z-1} = z^2 + z + 1, \text{ and} \\
\Phi_{15}(z) &= \tfrac{\Phi_3(z^5)}{\Phi_3(z)} = \tfrac{z^{10}+z^5+1}{z^2+z+1} \\
&= z^8 - z^7 + z^5 - z^4 + z^3 - z + 1, \text{ by (2.6a).} \\
\Phi_{75}(z) &= \Phi_{15}(z^5) \text{ by (2.6b),} \\
&= z^{40} - z^{35} + z^{25} - z^{20} + z^{15} - z^5 + 1. \\
\Phi_{150} &= \Phi_{75}(-z) \text{ by (2.7a),} \\
&= z^{40} + z^{35} - z^{25} - z^{20} + z^{15} + z^5 + 1
\end{aligned}$$

We formally describe this approach in algorithm 1.

Algorithm 1: Computing $\Phi_n(z)$ by repeated polynomial division

```
Input: $n = 2^{e_0} p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$, where $2 < p_1 < \cdots < p_k$,
       $e_0 \geq 0$, and $e_i > 0$ for $1 \leq i \leq k$
Output: $\Phi_n(z)$
1  $m \longleftarrow 1$
2  $\Phi_m(z) \longleftarrow z - 1$
3  for $i = 1$ to $k$ do
4      $\Phi_{mp_i}(z) \longleftarrow \Phi_m(z^{p_i})/\Phi_m(z)$          // By (2.6a)
5      $m \longleftarrow m \cdot p_i$
6  if $e_0 > 0$ then
7      $\Phi_{2m}(z) \longleftarrow \Phi_m(-z)$                          // By (2.7a)
8      $m \longleftarrow 2m$
   // $m$ is the largest squarefree divisor of $n$ now
9  $s \longleftarrow n/m$
10 $\Phi_n(z) \longleftarrow \Phi_m(z^s)$                               // By (2.6b)
   return $\Phi_n(z)$
```

While algorithm 1 is beautifully simple, it is not nearly the fastest way to compute $\Phi_n(z)$, particularly if we use classical polynomial division to calculate the polynomial quotient $\Phi_m(z^{p_i})/\Phi_m(z)$ (line 4). For even though the numerator is sparse, the denominator and quotient are typically dense.

We implemented algorithm 1 using the discrete fast Fourier transform (FFT) to perform $\Phi_m(z^{p_i})/\Phi_m(z)$ fast. This is done modulo suitably chosen primes q_j. The cost of computing one image of $\Phi_n(z)$ modulo a prime q is $\mathcal{O}(\phi(n)\log\phi(n))$ arithmetic operations in $\mathbb{Z}_q$. With each iteration of the loop on line 3, the degree of the resulting polynomial grows by a factor. As such the cost of this approach is dominated by the last division. For a description of the discrete FFT, we refer the reader to [3].

We compute images of $\Phi_n(z)$ modulo sufficiently many primes and recover the integer coefficients of $\Phi_n(z)$ using Chinese remaindering. In order to apply the FFT modulo q, we need a prime q with $2^k|q-1$ and $2^k > \phi(n)$, the degree of the output $\Phi_n(z)$. Since for $n > 10^9$ there are no such 32 bit primes, we used used 42-bit primes with arithmetic modulo q coded using 64-bit machine integer arithmetic.

It follows from lemma 1 that for primes $p|n$, the set of nonzero coefficients of $\Phi_{np}(z)$ and $\Phi_n(z)$ are the same. Similarly, by lemma 2 we have $A(n) = A(2n)$ for odd n. Thus $\Phi_n(z)$ for even or nonsquarefree n, for our purposes, are uninteresting. Moreover, if $\bar{n}$ is the largest odd squarefree divisor of n, then it is easy to obtain $\Phi_n(z)$ from $\Phi_{\bar{n}}(z)$.

For the remainder of this paper, we only consider $\Phi_n(z)$ for squarefree, odd n.

3. HIGH-PERFORMANCE ALGORITHMS FOR COMPUTING $\Phi_N(Z)$

Our C implementation of the FFT-based approach proved to be faster than methods available via computer algebra systems at the time. Using this method we were able to compute examples of $\Phi_n(z)$ of degree in the billions and height well beyond that. However, the FFT approach was eclipsed by a faster algorithm.

For $n > 1$, the number of squarefree divisors of n is even. As such we can rewrite (2.3) as

$$\Phi_n(z) = \prod_{d|n}(1 - z^d)^{\mu(\frac{n}{d})}. \tag{3.1}$$

As $\Phi_n(z)\Psi_n(z) = z^n - 1$ we also have, for $n > 1$,

$$\Psi_n(z) = -\prod_{d|n,d<n}(1 - z^d)^{-\mu(\frac{n}{d})}. \tag{3.2}$$

In our implementation of every algorithm presented in section 3, we compute $\Phi_n(z)$ as the product of terms $(1-z^d)^{\pm 1}$ appearing in (3.1); however, in the identities we present in this section it is often less cumbersome to express $\Phi_n(z)$ in terms of $(z^d-1)^{\pm 1}$. We refer to the $(1-z^d)^{\pm 1}$ (alternatively $(z^d - 1)^{\pm 1}$) comprising $\Phi_n(z)$ as the *subterms* of $\Phi_n(z)$.

Given that the power series expansion of $(1 - z^d)^{-1}$ is $(1 + z^d + z^{2d} + z^{3d} + \dots)$, it becomes equally easy to either multiply or divide by $(1 - z^d)$. $\Phi_n(z)$ can thus be computed as the truncated power series of

$$\prod_{\mu(\frac{n}{d})=1}(1 - z^d) \cdot \prod_{\mu(\frac{n}{d})=-1}(1 + z^d + z^{2d} + \dots), \tag{3.3}$$

as described in procedure SPS.

Procedure SPS(n), computing $\Phi_n(z)$ as a quotient of sparse power series

The Sparse Power Series (SPS) Algorithm

```
Input: $n$ a squarefree, odd integer
Output: $a(0), \dots, a(\frac{\phi(n)}{2})$, the first half of the
        coefficients of $\Phi_n(z)$
// we compute terms up to degree $D$
1 $D \longleftarrow \frac{\phi(n)}{2}$, $a(0) \longleftarrow 1$
2 for $1 \le i \le M$ do $a(i) \longleftarrow 0$
3 for $d|n$ such that $d < n$ do
4     if $\mu(\frac{n}{d}) = 1$ then              // multiply by $1 - z^d$
5         for $i = D$ down to $d$ by $-1$ do
6             $a(i) \longleftarrow a(i) - a(i - d)$
7     else                                     // divide by $1 - z^d$
8         for $i = d$ to $D$ do
9             $a(i) \longleftarrow a(i) + a(i - d)$
  return $a(0), a(1), \dots a(D)$
```

The brunt of the work of the SPS algorithm takes place on lines 6 and 9, where we multiply by $(1 - z^d)$ and $(1 - z^d)^{-1}$ respectively. In either case, computing this product, truncated to degree $D = \phi(n)/2$, takes $\mathcal{O}(D - d) \in \mathcal{O}(\phi(n))$ arithmetic operations in $\mathbb{Z}$. As n, a product of k distinct primes, has 2^k positive divisors, the SPS method requires some $\mathcal{O}(2^k \cdot \phi(n))$ operations to compute $\Phi_n(z)$ of order k. Note that while $1 - z^n$ appears in (3.1), we do not multiply by $1 - z^n$ is algorithm SPS, as it does not affect our result. This is because $1 - z^n \equiv 1 \pmod{z^D}$ for $D = \phi(n)/2$.

Using the analog identity for $\Psi_n(z)$, (3.2), we derive a very similar method for $\Psi_n(z)$, described by procedure SPS-Psi.

Procedure SPS-Psi(n), computing $\Psi_n(z)$ as a quotient of sparse power series

A Sparse Power Series Algorithm for $\Psi_n(z)$

```
Input: $n$ a squarefree, odd integer
Output: $b(0), \dots, b(\lfloor\frac{n-\phi(n)}{2}\rfloor)$, the first half of the
        coefficients of $\Psi_n(z)$
1 $D \longleftarrow \lfloor\frac{n-\phi(n)}{2}\rfloor$, $b(0) \longleftarrow 1$
2 for $1 \le i \le D$ do $b(i) \longleftarrow 0$
3 for $d|n$ such that $d < n$ do
4     if $\mu(\frac{n}{d}) = -1$ then            // multiply by $1 - z^d$
5         for $i = D$ down to $d$ by $-1$ do
6             $b(i) \longleftarrow b(i) - b(i - d)$
7     else                                     // divide by $1 - z^d$
8         for $i = d$ to $D$ do
9             $b(i) \longleftarrow b(i) + b(i - d)$
  return $b(0), b(1), \dots, b(D)$
```

By a similar analysis as for SPS, we see that procedure SPS-Psi requires $\mathcal{O}(2^k(n - \phi(n)) \in \mathcal{O}(2^k \cdot n)$ arithmetic operations.

3.1 The palindromic property of cyclotomic coefficients

In the SPS and SPS-Psi methods we truncate to half the degree of $\Phi_n(z)$ and $\Psi_n(z)$ respectfully. This is because it is trivial to obtain the ter,s of higher degree. For $n > 1$ the coefficients of $\Phi_n(z)$ are *palindromic* and those of $\Psi_n(z)$ are *antipalindromic*. That is, given

$$\Phi_n(z) = \sum_{i=0}^{\phi(n)} a(i)z^i \quad \text{and} \quad \Psi_n(z) = \sum_{i=0}^{n-\phi(n)} b(i)z^i,$$

we have that $a(i) = a(\phi(n) - i)$ and $b(i) = -b(n - \phi(n) - i)$. We prove a related result, which will bode useful in subsequent algorithms.

Lemma 3. *Let*

$$f(z) = \Phi_{n_1}(z) \cdot \Phi_{n_2}(z) \cdots \Phi_{n_s}(z) = \sum_{i=0}^{D} c(i)z^i \tag{3.4}$$

be a product of cyclotomic polynomials such that n_j is odd for $1 \le j \le s$. Then $c(i) = (-1)^D c(D - i)$ for $0 \le i < D$. In other words, if D is odd $f(z)$ is antipalindromic, and if D is even $f(z)$ is palindromic.

Proof. Clearly $f(z)$ is monic. If ω is a root of f, then ω is an $(n_j)_{th}$ primitive root of unity for some j such that $1 \le j \le s$. In which case, ω^{-1} is also an $(n_j)_{th}$ primitive root of unity and hence is also a root of $f(z)$. Set

$$g(z) = z^D f(z^{-1}) = \sum_{i=0}^{D} c(D - i)(z). \tag{3.5}$$

$g(z)$ is a polynomial of degree D with leading coefficient $c(0)$ whose roots are the roots of f. Thus $f(z)$ and $g(z)$ only differ by the constant factor $c(0)/c(D) = c(0)$. We need only resolve $c(0)$. To that end, we observe that $\phi(n)$ is even for odd $n > 1$, and $\phi(1) = 1$. Thus $r \equiv D \pmod 2$, where r is the cardinality of

$$\{j : 1 \le j \le s \text{ and } n_j = 1\}. \tag{3.6}$$

The constant term of f, $c(0)$, is the product of the constant terms of the $\Phi_{n_j}(z)$ in (3.4). Since the constant term of $\Phi_1(z) = z - 1$ is -1, and by (3.1), the constant term of $\Phi_n(z)$ is 1 for $n > 1$, we have that $c(0) = (-1)^r = (-1)^D$, completing the proof. $\square$

We note that lemma 3 does not hold if we relax the restriction that n_j must be odd in (3.4). Consider the trivial counterexample $\Phi_2(z) = z+1$. By (2.5), we have that $\Psi_n(z)$ is a product of cyclotomic polynomials, and so lemma 3 applies to $\Psi_n(z)$ for odd n, or any product of the form

$$\Psi_{n_1}(z) \cdot \Psi_{n_2}(z) \cdots \Psi_{n_s}(z), \tag{3.7}$$

where $n_1, n_2, \dots, n_s$ are all odd.

3.2 Improving the sparse power series method by further truncating degree

The sparse power series algorithm slows appreciably as we calculate $\Phi_n(z)$ for n with increasingly many factors. The slowdown in computing $\Phi_{np}(z)$ compared to $\Phi_n(z)$ is twofold. By introducing a new prime factor p we double the number of subterms $(1-z^d)^{\pm 1}$ in our product (3.1). In addition, the degree of $\Phi_{np}(z)$ is $p-1$ times that of $\Phi_n(z)$, thus increasing the cost of multiplying one subterm $(1-z^d)^{\pm 1}$ by a factor. For $\Phi_n(z)$ of larger degree the algorithm also exhibits poorer locality.

In procedure SPS, we effectively compute 2^k distinct power series, each a product of subterms $(1-z^d)^{\pm 1}$, each truncated to degree $\phi(n)/2$. We can improve the SPS algorithm if we truncate any intermediate power series to as minimal degree necessary, thereby reducing the number of arithmetic operations and leveraging locality where possible. We let the *degree bound* refer to the degree we must truncate to at some stage in the computation of $\Phi_n(z)$ using the SPS algorithm or a variant thereof.

Depending on the order in which we multiply the subterms of $\Phi_n(z)$, some of the intermediate products of subterms we compute may be polynomials as well. If, at some point of our computation of $\Phi_n(z)$, we have a product of subterms that is a polynomial $f(z)$ of degree D, then $f(z)$ is a product of cyclotomic polynomials satisfying lemma 3 (provided n is odd and squarefree), and we need only truncate to degree at most $\lfloor D/2 \rfloor$ at previous stages of the computation.

Once we have computed $f(z)$, our degree bound may increase. In which case we can generate higher-degree terms of $f(z)$ as necessary using lemma 3.

More generally, if we have some product of subterms of $\Phi_n(z)$ and we know polynomials $f_1(z), f_2(z), \dots f_s(z)$ of degrees $D_1, D_2, \dots, D_s$ will occur as products of subterms at later stages of our computation, then we can truncate to $\lfloor D/2 \rfloor$, where $D = \min_{1 \le j \le s} D_s$. Our aim is to order the subterms in an intelligent manner which minimizes the growth of the degree bound over the computation of $\Phi_n(z)$.

To further our aim, we let $n = mp$, where p is the largest prime divisor of n and $m > 1$. In which case

$$\begin{aligned}\Phi_{mp}(z) =& \frac{\Phi_m(z^p)}{\Phi_m(z)} \text{ by lemma 1,} \\ =& \Psi_m(z) \cdot \Phi_m(z^p) \cdot (z^m - 1)^{-1}.\end{aligned} \tag{3.8}$$

By (3.1) and (3.2), we can break $\Psi_m(z)$ and $\Phi_m(z)$ into respective products of subterms.

$$\Phi_n(z) = \left(\prod_{d|m, d<m} (z^d - 1)^{-\mu(\frac{m}{d})} \right) \left(\prod_{d|m} (z^{dp} - 1)^{\mu(\frac{m}{d})} \right) (z^m - 1)^{-1}. \tag{3.9}$$

Thus to compute $\Phi_n(z)$, we can first compute $\Psi_m(z)$, the leftmost product of (3.9) to degree $\frac{m-\phi(m)}{2}$, use the antipalindromic property of $\Psi_m(z)$ to reconstruct its remaining coefficients, and then multiply the remaining subterms as we would in algorithm SPS. Algorithm SPS2 describes the method.

Procedure SPS2(n) : First revision of algorithm **SPS**

Algorithm SPS2: Improved Sparse Power Series

```
Input: $n = mp$, a squarefree, odd integer with greatest
       prime divisor $p$
Output: $a(0), \dots, a(\frac{\phi(n)}{2})$, the first half of the
        coefficients of $\Phi_n(z)$
// Compute first half of $\Psi_m(z)$
1  $a(0), a(1), \dots, a(\lfloor \frac{n-\phi(m)}{2} \rfloor) \longleftarrow$ SPS-Psi($m$)

// Construct other half of $\Psi_m(z)$ using lemma 3
2  $D \longleftarrow m - \phi(m)$
3  for $i = \lceil \frac{m-\phi(m)}{2} \rceil$ to $D$ do $a(i) \longleftarrow -a(m - \phi(m) - i)$

// Multiply by $\Phi_m(z^p)$
4  $D \longleftarrow \frac{\phi(n)}{2}$
5  $a(m - \phi(m) + 1), a(m - \phi(m) + 2), \dots, a(D) \longleftarrow 0$
6  for $d|m$ do
7      if $\mu(\frac{n}{d}) = 1$ then
8          for $i = D$ down to $d$ by $-1$ do
9              $a(i) \longleftarrow a(i) - a(i - dp)$
10     else
11         for $i = d$ to $D$ do
12             $a(i) \longleftarrow a(i) + a(i - dp)$

// Divide by $z^m - 1 = (-1 - z^m - z^{2m} - \dots)$
13 for $i = m$ to $D$ do $a(i) \longleftarrow -a(i) - a(i - m)$
   return $a(0), a(1), \dots a(D)$
```

For $n = mp$ with k distinct prime divisors, $\Psi_m(z)$ comprises $2^{k-1} - 1$ of the 2^k subterms of $\Phi_n(z)$. For each of these subterms appearing in $\Psi_m(z)$ we truncate to degree $(m - \phi(m))/2$. The asymptotic operation cost of SPS2 is no different than that of SPS; however, in practise this method cuts the running time in half (see table 1 for timings).

We note that the speed-up is not as substantial for m with very few prime factors. In the event that n is prime, we have $m = 1$ and $\Psi_m(z) = 1$. In such case the execution of SPS and SPS2 are effectively the same. For $n = qp$, a product of two primes with $q < p$, $\Phi_n(z)$ has only four subterms and we only get gains on the single subterm appearing in $\Psi_q(z) = z - 1$. For $\Phi_n(z)$ of low order, the proportion of subterms of $\Phi_n(z)$ appearing in $\Psi_m(z)$ is further from $1/2$ compared to $\Phi_n(z)$ for highly composite n (i.e. n for which k is larger). That said, however, $\Phi_n(z)$ is already easy to compute by the original SPS method for $\Phi_n(z)$ of low order, as these cyclotomic polynomials have very few subterms.

3.3 Calculating $\Phi_n(z)$ by way of a product of inverse cyclotomic polynomials

In algorithm SPS2 we bound to a smaller degree than in SPS when multiplying the first $2^{k-1} - 1$ subterms of $\Phi_n(z)$. We are able to lower the degree bound for many of the remaining $2^{k-1}+1$ subterms of $\Phi_n(z)$. To that end we establish the next identity.

Let $n = p_1p_2 \cdots p_k$, a product of k distinct odd primes. For $1 \leq i \leq k$, let $m_i = p_1p_2 \cdots p_{i-1}$ and $e_i = p_{i+1} \cdots p_k$. We set $m_1 = e_k = 1$, and let $e_0 = n$. Note that $n = e_ip_in_i$ for $1 \leq i \leq k$. In addition, $e_{i-1} = p_ie_i$ and $m_{i+1} = m_ip_i$. We restate (3.8), which was key to SPS2, as

$$\Phi_n(z) = \frac{\Psi_{m_k}(z^{e_k})}{(z^{n/p_k} - 1)} \Phi_{m_k}(z^{e_{k-1}}). \tag{3.10}$$

By repeated application of lemma 1, we have

$$\begin{aligned}\Phi_n(z) &= \frac{\Psi_{m_k}(z^{e_k})}{(z^{n/p_k} - 1)} \frac{\Psi_{m_{k-1}}(z^{e_{k-1}})}{(z^{n/p_{k-1}} - 1)} \Phi_{m_{k-1}}(z^{e_{k-2}}), \\ &\cdots \\ &= \frac{\Psi_{m_k}(z^{e_k})}{(z^{n/p_k} - 1)} \cdots \frac{\Psi_{m_2}(z^{e_2})}{(z^{n/p_2} - 1)} \frac{\Psi_{m_1}(z^{e_1})}{(z^{n/p_1} - 1)} \Phi_{m_1}(z^{e_0}), \\ &= \left(\prod_{j=1}^{k} \frac{\Psi_{m_j}(z^{e_j})}{(z^{n/p_j} - 1)} \right) \cdot \Phi_{m_1}(z^{e_0}).\end{aligned} \tag{3.11}$$

As $\Psi_{m_1}(z^{e_1}) = \Psi_1(z^{e_1}) = -1$, and $\Phi_1(z^{e_0}) = \Phi_1(z^n) = z^n - 1$, this simplifies to

$$\Phi_n(z) = \prod_{j=2}^{k} \Psi_{m_j}(z^{e_j}) \cdot \prod_{j=1}^{k} (z^{n/p_j} - 1)^{-1} \cdot (z^n - 1) \tag{3.12}$$

For example, for $n = 105 = 3 \cdot 5 \cdot 7$,

$$\Phi_{105}(z) = \Psi_{15}(z)\Psi_3(z^7)(z^{15} - 1)^{-1}(z^{21} - 1)^{-1}(z^{35} - 1)^{-1}(z^{105} - 1)$$

As with algorithm SPS2, we first calculate half the terms of $\Psi_{m_k}(z^{e_k}) = \Phi_{p_1p_2 \cdots p_{k-1}}(z)$, those with degree at most $\lfloor \frac{\phi(m_k)}{2} \rfloor$. We then iteratively compute the product

$$\Psi_{m_k}(z^{e_k}) \cdots \Psi_{m_2}(z^{e_2}) \tag{3.13}$$

from left to right. When calculating the degree of $\Psi_{m_j}(z^{e_j})$ we truncate to degree at most

$$\left\lfloor \frac{1}{2} \prod_{i=j}^{k} (m_i - \phi(m_i))e_i \right\rfloor, \tag{3.14}$$

half the degree of the product in (3.13). As our intermediate product grows larger we have to truncate to larger degree. The term $\Psi_{m_i}(z^{e_i})$, comprises $2^{i-1} - 1$ subterms of $\Phi_n(z)$. We compute $\Psi_{m_k}(z^{e_k})$ first because that contains 2^{k-1} subterms, nearly half of the 2^k we must multiply by to compute $\Phi_n(z)$, so it is best that we multiply these subterms first before the degree bound swells.

We leverage lemma 3 again when computing the product (3.13). Suppose we have half the terms of

$$f(z) = \prod_{i=j+1}^{k} \Psi_{m_i}(z^{e_i}),$$

for some $j \geq 2$ and we want to compute

$$g(z) = f(z) \cdot \Psi_{m_j}(z^{e_j}),$$

towards the aim of obtaining $\Phi_n(z)$. As both $f(z)$ and $g(z)$ have the (anti)palindromic property of lemma 3, when computing $g(z)$ we need to truncate to degree at most $\lfloor D/2 \rfloor$, where D is the lesser of

$$D_g = \prod_{i=j-1}^{k} (m_i - \phi(m_i))e_i \qquad \text{and} \qquad \phi(n),$$

the former of which is the degree of $g(z)$, the latter being the degree of $\Phi_n(z)$. Thus we apply lemma 3 to generate the higher-degree terms of $f(z)$ up to degree D. Once we have the product (3.13) we then apply the palindromic property again to generate the coefficients up to degree $\phi(n)/2$, provided we do not have them already. We then divide by the subterms $(1 - z^{n/p_j})$ for $1 \leq j \leq k$, truncating, again, to degree $\phi(n)/2$. We describe this approach in procedure SPS3. We assume the e_i and m_i were precomputed.

For n a product of one or two primes, SPS3 executes the same as in SPS2, and we see no gains. We only begin to see improved performance for n a product of three primes. In practise, we see the biggest improvement in performance when computing $\Phi_n(z)$ with many distinct prime factors. These are the cyclotomic polynomials which are most difficult to compute.

We do not have an intelligible analysis of the asymptotic operation cost of algorithm SPS3. We try to answer, however, for what subterms of $\Phi_n(z)$ do we truncate to lower degree using SPS3 versus SPS2? For the $2^{k-1} - 1$ subterms appearing in $\Psi_{m_k}(z^{e_k})$ we truncate to the same degree as in SPS2. These are exactly the subterms for which SPS2 had gains over SPS. For the k subterms of the form $(1 - z^{n/p})$, we truncate to degree $\phi(n)/2$ in SPS3. Moreover, the degree of the product in (3.13) is, by (3.12),

$$\phi(n) - n + \sum_{p|n} n/p. \tag{3.15}$$

Thus (3.13) potentially has degree greater than that of $\Phi_n(z)$, provided

$$1/p_1 + 1/p_2 + \cdots + 1/p_k > 1. \tag{3.16}$$

So, for some n there may exist additional subterms for which we do not have gains. For $n = p_1p_2 \cdots p_k$ for which $\Phi_n(z)$ is presently feasible to compute, however, it is seldom the case that (3.16) holds. The smallest odd, squarefree n for which (3.16) holds is $n = 3,234,846,615$, the product of the first nine odd primes. Thus for n a product of $k < 9$ distinct primes we have gains for all the remaining subterms. In any case, we always truncate to a lower degree than in procedure SPS2 when calculating $\Psi_{m_i}(z^{e_i})$ for $k-8 < i < k$. As $\Psi_{m_k}(z^{e_k}) \cdots \Psi_{m_{k-7}}(z^{e_{k-7}})$ comprise $2^{k-1} - 2^{k-8} - 8$, or close to half of the 2^k subterms.

Quantifying these gains is more difficult. Timings suggest, however, that for n with 6 or more factors, computing $\Phi_n(z)$

Procedure SPS3(n) : Second revision of algorithm **SPS**

```
Algorithm SPS3: Iterative Sparse Power Series

Input: $n = p_1 p_2 \dots p_k$, a squarefree product of $k$ primes
Output: $a(0), \dots, a(\frac{\phi(n)}{2})$, the first half of the
        coefficients of $\Phi_n(z)$
$a(0), a(1), a(2), \dots, a(\phi(n)/2) \longleftarrow 1, 0, 0, \dots, 0$
$D_f \longleftarrow 0$, $D_g \longleftarrow m_k - \phi(m_k)$, $D \longleftarrow \min(D_g, \phi(n))$
for $j = k$ down to 2 do
    // $\times$ by $\Psi_{m_j}(z^{e_j})$; truncate to degree $\lfloor D/2 \rfloor$
    for $d | m_j$ such that $d < m_j$ do
        if $\mu(\frac{n}{d}) = -1$ then
            for $i = D$ down to $d$ by $-1$ do
                $a(i) \longleftarrow a(i) - a(i-d)$
        else
            for $i = d$ to $D$ do
                $a(i) \longleftarrow a(i) + a(i-d)$
    $D_f \longleftarrow D_g$
    if $j > 2$ then
        $D_g \longleftarrow D_g + (m_{j+1} - \phi(m_{j+1}))e_{j+1}$
        $D \longleftarrow \min(D_g, \phi(n))$
    else $D \longleftarrow \phi(n)$
    // Use lemma 3 to get higher-degree terms
    for $i \longleftarrow \lfloor D_f/2 \rfloor + 1$ to $\lfloor D/2 \rfloor$ do
        $a(i) \longleftarrow (-1)^{D_f} a(D_f - i)$

// $\div$ by $(1 - z^{n/p_j})$; truncate to degree $\phi(n)/2$
for $j = 1$ to $k$ do
    for $i = n/p_j$ to $\phi(n)/2$ do
        $a(i) \longleftarrow a(i) + a(i - n/p_j)$
return $a(0), a(1), \dots, a(\phi(n)/2)$
```

using SPS3 is between 3 and 5 times faster than SPS2 (see section 4). The speed-up is typically larger for n with more prime factors.

3.4 Calculating $\Phi_n(z)$ and $\Psi_n(z)$ recursively.

Algorithm SPS3 depended on the identity (3.12), which describes $\Phi_n(z)$ in terms of a product of inverse cyclotomic polynomials of decreasing order and index. We derive a similar expression for $\Psi_n(z)$. Let m_i and e_i be as defined in section 3.3, and again let $n = p_1 p_2 \cdots p_k$ be a product of k distinct odd primes where $p_1 < p_2 < \dots p_k$. Again by repeated application of lemma 1,

$$\begin{aligned} \Psi_n(z) &= \Phi_{m_k}(z^{e_k})\Psi_{m_k}(z^{e_{k-1}}), \\ &= \Phi_{m_k}(z^{e_k})\Phi_{m_{k-1}}(z^{e_{k-1}})\Psi_{m_{k-1}}(z^{e_{k-2}}), \\ &\dots \\ &= \Phi_{m_k}(z^{e_k}) \cdots \Phi_{m_1}(z^{e_1})\Psi_{m_1}(z^{e_1}). \end{aligned} \tag{3.17}$$

As $m_1 = 1$ and $\Psi_1(z) = 1$, we thus have that

$$\Psi_n(z) = \prod_{j=1}^{k} \Phi_{m_j}(z^{e_j}). \tag{3.18}$$

(3.12) and (3.18) suggest a recursive method of computing $\Phi_n(z)$. Consider the example of $\Phi_n(z)$, for $n = 1155 = 3 \cdot 5 \cdot 7 \cdot 11$. To obtain the coefficients of $\Phi_{1105}(z)$, procedure SPS3 constructs the product

$$\begin{aligned} &\Psi_{105}(z)\Psi_{15}(z^{11})\Psi_3(z^{77})(1 - z^{385})^{-1} \cdot \\ &\cdot (1 - z^{231})^{-1}(1 - z^{165})^{-1}(1 - z^{105})^{-1}(1 - z^{1155}) \end{aligned} \tag{3.19}$$

from left to right. However, in light of (3.18), we know this method computes $\Psi_{105}(z)$ in a wasteful manner. We can treat $\Psi_{105}(z)$ as a product of cyclotomic polynomials of smaller index:

$$\Psi_{105}(z) = \Phi_{15}(z)\Phi_5(z^7)\Phi_1(z^{35}).$$

One could apply (3.12) yet again, now to $\Phi_{15}(z)$, giving us

$$\Phi_{15}(z) = \Psi_5(z)(1 - z^5)^{-1}(1 - z^3)^{-1}(1 - z^{15}).$$

Upon computing $\Psi_{105}(z)$, we can break the next term of (3.19), $\Psi_{15}(z^{11})$ into smaller products in a similar fashion. We effectively compute $\Phi_n(z)$ by recursion into the factors of n. We call this approach the recursive sparse power series method, and we describe our implemetation in procedure SPS4.

SPS4 effectively takes a product of cyclotomic polynomials $f(z)$, and multiplies by either $\Phi_m(z^e)$ (or $\Psi_m(z^e)$), by recursion described above. If we are to multiply by $\Psi_m(z^e)$, upon completion of our last recursive call, we are finished (line 8 of SPS4). This is because $\Psi_m(z^e)$ is exactly a product of cyclotomic polynomials by (3.18). If, however, we are to multiply by $\Phi_m(z^e)$, once we have completed our last recursive call, we need to divide and multiply by some additional subterms (lines 10 and 13), as is necessary by the identity (3.13).

Obtaining the degree bound in the recursive SPS method is not as immediate as in the previous SPS algorithms. In the iterative SPS our algorithm produces a sequence of intermediate polynomials. With the possible exception that the output polynomial $\Phi_n(z)$, these polynomials are in order of increasing degree. In the recursive SPS, however, we no longer have this monotonic property.

The difference between the degree bound in the iterative SPS and the recursive SPS, is that in the former we truncate to the least degree of two polynomials, whereas in the recursive sparse power series case, we may bound by the least degree of many polynomials. Moreover, we need to know what degree to bound to at each level of recursion. Procedure SPS4 has an additional parameter, D, which serves as a bound on the degree.

As before, let $f(z)$ be a product of cyclotomic polynomials. Let D_f be the degree of $f(z)$ and suppose, while we are in some intermediate step of the computation of $\Phi_n(z)$ or $\Psi_n(z)$, that we have the first $\lfloor D_f/2 \rfloor + 1$ terms of $f(z)$, and we want next to compute the terms of

$$g(z) = f(z) \cdot \Phi_m(z^e) \qquad (\text{or } f(z) \cdot \Phi_m(z^e)), \tag{3.20}$$

up to degree $\lfloor D/2 \rfloor$, for some $D \in \mathbb{N}$. D is effectively the degree of some product of cyclotomic polynomials we will eventually obtain later at some previous level of recursion. If we let D_g be the degree of $g(z)$, then when computing $g(z)$ from $f(z)$ we need only compute terms up to $\lfloor D^*/2 \rfloor$, where $D^* = \min(D, D_g)$ (line 3). Thus when we recurse in SPS4, if $D_g < D$ we lower the degree bound from D to D_g.

To guarantee that we can obtain higher-degree terms whenever necessary we impose the following rule: If SPS4 is given $f(z)$ and is to output $g(z)$, we require that $f(z)$ is truncated to degree $\lfloor D'/2 \rfloor$ on input, where $D' = \min(D, D_f)$, and

Procedure SPS4(m, e, λ, D_f, D, a) : Multiply a product of cylotomic polynomials by $\Phi_m(z^e)$ or $\Psi_m(z^e)$

SPS4: A Recursive Sparse Power Series Algorithm.
Input:

- m, a positive, squarefree odd integer; λ, a boolean; $D \in \mathbb{Z}$, a bound on the degree
- D_f, the degree of $f(z)$, a product of cyclotomic polynomials partially stored in array a. D_f is passed by value.
- An array of integers $a = [a(0), a(1), \dots]$, for which, given $f(z)$, $a(0)$,$a(1), \dots, a(\lfloor D'/2 \rfloor)$ are the first $\lfloor D'/2 \rfloor + 1$ coefficients of f, where $D' = min(D_f, D)$. a is passed by reference.

Result:
If λ is true, we compute $g(z) = f(z)\Phi_m(z^e)$, otherwise, we compute $g(z) = f(z)\Psi_m(z^e)$. In either case we truncate the result to degree $\lfloor D^*/2 \rfloor$, where $D^* = min(D, D_g)$. We write the coefficients of g to array a, and return the degree of g, D_g.

```
1  if $\lambda$ then $D_f \longleftarrow D + \phi(m)e$
2  else $D_f \longleftarrow D + (m - \phi(m))e$

3  $D^* \longleftarrow \min(D_g, D)$   // $D^*$ is our new degree bound
4  $e^* \longleftarrow e$, $m^* \longleftarrow m$, $D^* \longleftarrow D$

5  while $m^* > 1$ do
6      $p \longleftarrow$ (largest prime divisor of $m^*$), $m^* \longleftarrow m/p$
7      $D_f \longleftarrow$ SPS4($m^*$, $e^*$, not $\lambda$, $D_f$, $D^*$, $a$), $e^* \longleftarrow e^*p$
8  if not $\lambda$ then      // We have multiplied by $\Psi_m(z^e)$
       return $D_g$

   // Get higher degree terms as needed
9  for $\lfloor D_f/2 \rfloor + 1$ to $\lfloor D^*/2 \rfloor$ do $a(i) \longleftarrow (-1)^{D_f} a(D_f - i)$

   // Divide by $(1 - z^{me/p})$ for $p|m$
10 for each prime $p|m$ do
11     for $i = (me/p)$ to $\lfloor D^*/2 \rfloor$ do
12         $a(i) \longleftarrow a(i) + a(i - me/p)$

   // multiply by $1 - z^{me}$
13 for $i = \lfloor D^*/2 \rfloor$ down to $d$ do
14     $a(i) \longleftarrow a(i) - a(i - me)$
   return $D_g$
```

that $g(z)$ is truncated to degree $\lfloor D^*/2 \rfloor$ on output. Note that the degree bound on $g(z)$ is always at least the bound on $f(z)$; it will only increase over the computation of $\Phi_n(z)$.

To calculate the first half of the coefficients of $\Phi_n(z)$, one would merely set

$$(a(0), a(1), a(2), \dots, a(\phi(n)/2) = (1, 0, 0, \dots, 0)$$

and call SPS4(n,1,true,0,$\phi(n)$, a)). Similarly, to calculate the first half of $\Psi_n(z)$ we would call SPS4(n,1,false,0,$n-\phi(n)$,a).

3.4.1 Implementing the recursive SPS algorithm

In procedure SPS4 we often need the prime divisors of input m. It is obviously wasteful to factor m every time we recurse. To compute $\Phi_n(z)$ or $\Psi_n(z)$ for squarefree n, we first precompute the factorization of n and store it in a global array $P = [p_1, p_2, \dots, p_k]$. Upon calling SPS4, every subsequent recursive call will multiply by some (inverse) cyclotomic polynomial of index $m|n$. Our implementation of the recursive sparse power series algorithm has an additional argument, $B = [b_1, b_2, \dots, b_k]$, a series of bits, that, given P, gives us the factorization of m. We set b_i to 1 if p_i divides m, and zero otherwise. For all tractable cases, B can fit in two bytes and in most practical cases, one byte.

Thus, in the while loop on line 5 in SPS4, we take a copy of B, call it B^*, and scan it for nonzero bits. Each time we find a nonzero bit we set that bit to zero, and pass B^* by value to the recursive call occuring on line 7 of procedure SPS4. We continue in this fashion until all the bits of B^* are set to zero. We similarly scan the bits of B again to later obtain the prime divisors of n, as is needed on line 10 of procedure SPS4.

We find that the recursive SPS is slightly faster than the iterative SPS; however, this improvement is not nearly as substantial as was the iterative SPS over prior versions. While the degree bound computing $\Phi_n(z)$ with the recursive SPS is always less than or equal to that using the iterative SPS, the recursive structure of the program results in additional overhead. We could program the recursive SPS iteratively; however, we would effectively have to create our own stack to mimic recursion.

4. PERFORMANCE AND TIMINGS

We first provide a visual comparison of the SPS algorithms computing explicit examples of $\Phi_n(z)$. Figures 1 and 2 show how the degree bound grows in algorithms SPS1-4 over the computation of $\Phi_n(z)$ for

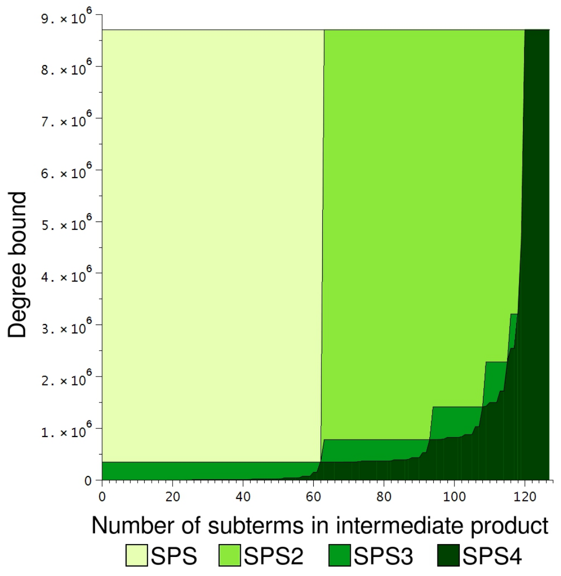
$$n = 43730115 = 3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 37 \text{ and}$$
$$n = 3234846615 = 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29.$$

respectively. In both figures, the horizontal axis represents how many subterms we have in our intermediate product. As $n = 43730115$ has 7 unique prime divisors, there are some $2^7 - 1 = 127$ intermediate products of subterms produced over the computation, excluding the final result $\Phi_n(z)$. As the computation of $\Phi_n(z)$ progresses we traverse from left to right in figures 1 and 2, and the degree bound increases.

We should note that in SPS2-4, the degree bound in each is at most the degree bound of its predecessor. In figure 1, we associate the darkest green region with SPS4; the two darkest green regions with SPS3; the three darkest green regions with SPS2; and all four green areas represent the degree bound for SPS. The height of the regions associated with a version of the SPS algorithm represents its degree bound at that stage of the computation. Figure 2, in red, should be interpreted similarly. In the case of SPS the degree bound is always the constant $\phi(n)/2$.

We think of the area of the regions in figure 1 associated with a version of the SPS algorithm as a heuristic measure of its time cost. One could think of the savings of SPS4 over SPS3, for instance, as the area of the second darkest green region. The area of the three darker green regions is slightly over half the area of all four. As such, we expect that SPS2 would take roughly half as much time as SPS. Moreover, by this measure we expect that SPS3 should be considerably faster than SPS2, and SPS4 should be marginally faster than SPS3. This is comparable with our timings in table 1. The

Figure 1: Growth of the degree bound over the computation of $\Phi_{43730115}(z)$ using SPS1-4



degree bounds in figure 2 show a similar, albeit more clearly defined shape.

We timed our implementations on a system with a 2.67GHz Intel Core i7 quad-core processor and 6 GB of memory. All of our aglorithms are implemented in C and are single-threaded. Here we time our 64-bit precision implementations of procedures SPS1-4, each of which check for integer overflow using inline assembly. Our implementation of algorithm 1 calculates $\Phi_n(z)$ modulo two 32-bit primes and reconstructs $\Phi_n(z)$ by Chinese remaindering.

Table 1: Time to calculate $\Phi_n(z)$ (in seconds*)

	algorithm				
n	FFT	SPS	SPS2	SPS3	SPS4
255255	0.40	0.00	0.00	0.00	0.00
1181895	1.76	0.01	0.00	0.00	0.00
4849845	7.74	0.12	0.06	0.02	0.01
37182145	142.37	1.75	0.95	0.23	0.19
43730115	140.62	1.69	0.93	0.23	0.19
111546435	295.19	6.94	3.88	1.45	0.94
1078282205	-	105.61	58.25	12.34	9.29
3234846615	-	432.28	244.44	81.32	49.18

*times are rounded to the nearest hundredth of a second

As the number of distinct prime factors of n plays a significant role in the cost of computing $\Phi_n(z)$, we list the factors of n (table 2) and $A(n)$ (table 3) for n appearing in table 1.

For the SPS and SPS4 algorithms, we have implemented, in addition to the 64-bit version, 8-bit, 32-bit, and 128-bit precision versions. We do not use GMP multiprecision integer arithmetic. It was easy to implement multiprecision arithmetic for our specific purpose as we only add and subtract coefficients in the SPS algorithms. We also have a version of SPS and SPS4 which calculates images of $\Phi_n(z)$ modulo 32-bit primes, writes the images to the harddisk, and then reconstruct $\Phi_n(z)$ from the images by way of Chinese remaindering. This implementation is most useful for $\Phi_n(z)$

Figure 2: Growth of the degree bound over the computation of $\Phi_{3234846615}(z)$ using SPS1-4

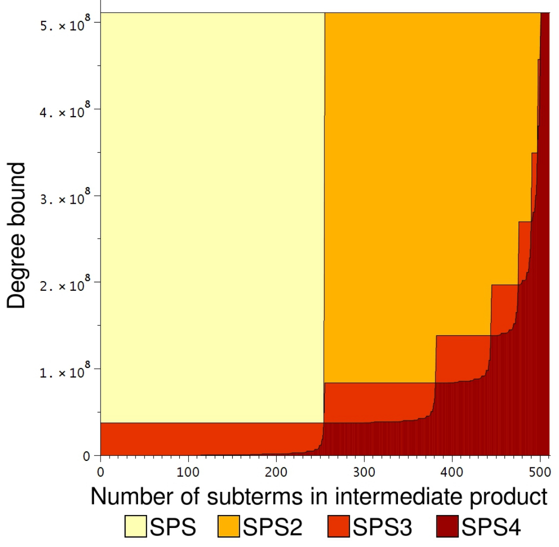


Table 2: Factorization of n, for n from table 1

n	factorization of n
255255	$3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17$
1181895	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29$
4849845	$3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19$
37182145	$5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$
43730115	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 37$
111546435	$3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$
1078282205	$5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29$
3234846615	$3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29$

Table 3: $A(n)$ for n from table 1

n	height $A(n)$
255255	532
1181895	14102773
4849845	669606
37182145	2286541988726
43730115	862550638890874931
111546435	1558645698271916
1078282205	8161018310
3234846615	2888582082500892851

which we cannot otherwise fit in main memory.

5. CURRENT WORK

We have implemented the algorithms in this paper to create a library of data on the heights and lengths of cyclotomic polynomials. This data is available at

`http://www.cecm.sfu.ca/~ada26/cyclotomic/`

A 64-bit implementation of the SPS4 algorithm, written in C but without overflow check, is also made available at the website.

We aim to compute the coefficients of $\Phi_n(z)$, for

$$n = 3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 37 \cdot 43 \cdot 53 = 99660932085.$$

We expect that this cyclotomic polynomial will have very large height. We have previously verified that

$$A(\tfrac{n}{53}) = 645409970360109115668264461815238889715563 \text{ and}$$
$$A(\tfrac{n}{43}) = 670759626669230198236020306631531188O3367$$

are the smallest two examples of k such that $A(k) > k^4$. Both $A(\frac{n}{53})$ and $A(\frac{n}{43})$ are greater than 2^{135}.

We previously attempted to compute $\Phi_n(z)$ using an implementation of the SPS algorithm. We computed images of $\Phi_n(z)$ modulo 32-bit primes. Storing half of $\Phi_n(z)$ to 32-bit precision takes roughly 76 GB of space. We do not have enough RAM to store these images in main memory, so we read and wrote intermediate results to the hard disk. This proved to be slow, as each image required us to make $2^9 = 512$ passes over the hard disk. We computed four images of $\Phi_n(z)$, after which the hard disk crashed.

In light of the development of the new variants of the SPS algorithms, we have a new approach to compute $\Phi_n(z)$. We want to minimize hard disk reads and writes. This is because performing the computation on the harddisk is appreciably slower and potentially more error-prone than in memory. We are limited to 16 GB of RAM. We expect that $A(n) < 2^{320}$; that is, 320-bit precision will be sufficient to construct $\Phi_n(z)$. Towards our aim, let

$$f(z) = \Psi_{m_9}(z)\Psi_{m_8}(z^{53}) \tag{5.1}$$

where $m_9 = \frac{n}{53} = 1,880,394,945$ and
$m_8 = \frac{n}{43\cdot 53} = 43,730,115$. By (3.11), we have

$$\Phi_n(z) = f(z)(1 - z^{n/53})^{-1}(1 - z^{n/43})^{-1}\Phi_{m_8}(z^{43\cdot 53}). \tag{5.2}$$

$f(z)$ has degree less than $2.55 \cdot 10^9$. We can compute images of $f(z)$ modulo 64-bit primes using roughly 10 GB of RAM, then extract $f(z)$ from its images by way of Chinese remaindering. After which we will compute the coefficients of the truncated power series

$$\begin{aligned} g(z) &= \sum_{i=0}^{\phi(n)/2} c(i)z^i, \\ &= f(z)(1 - z^{n/53})^{-1}(1 - z^{n/43})^{-1} \bmod z^{\phi(n)/2+1}. \end{aligned} \tag{5.3}$$

This will entail two passes over the hard disk, one per division by $1 - z^{n/53}$ or $1 - z^{n/43}$. We produce the coefficients of $g(z)$ in order of ascending degree during the second pass of the harddisk. Storing $g(z)$ or $\Phi_n(z)$ at this precision up to degree $\phi(n)/2$ requires more than 750 GB of storage. We can reorganize the terms of $g(z)$ in a manner which allows us to compute the coefficients of $\Phi_n(z)$ in memory. For $0 \le j < 43 \cdot 53 = 2279$, let

$$g_j(z) = \sum_{0 \le i\cdot 2279+j \le \phi(n)/2} c(i)z^i \tag{5.4}$$

We can construct the $g_j(z)$ as we sequentially produce the terms of $g(z)$. We have that

$$g(z) = \sum_{j=0}^{2278} z^j \cdot g_j(z^{2279}),$$

and thus by (5.2),

$$\Phi_n(z) \equiv \sum_{j=0}^{2278} z^j \cdot g_j(z^{2279})\Phi_{m_8}(z^{2279}) \pmod{z^{\phi(n)/2+1}}.$$

Thus to produce the first half of the coefficients of $\Phi_n(z)$, it suffices to compute $g_j(z) \cdot \Phi_{m_8}(z)$, for $0 \le j < 2279$. Each polynomial has degree less than $2.6 \cdot 10^6$, and can be computed to 320-bit precision with less than a GB of memory.

6. REFERENCES

[1] A. Arnold and M. Monagan. Calculating cyclotomic polynomials. Submitted to *Mathematics of Computation*, available at `http://www.cecm.sfu.ca/~ada26/cyclotomic/`.

[2] P. Erdős. On the coefficients of the cyclotomic polynomial. *Bull. Amer. Math. Soc.*, 52:179–184, 1946.

[3] K.O. Geddes, S.R. Czapor, and G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, Boston, 1992.

[4] Y. Koshiba. On the calculations of the coefficients of the cyclotomic polynomials. *Rep. Fac. Sci. Kagoshima Univ.*, (31):31–44, 1998.

[5] Y. Koshiba. On the calculations of the coefficients of the cyclotomic polynomials. II. *Rep. Fac. Sci. Kagoshima Univ.*, (33):55–59, 2000.

[6] T.D. Noe. Personal communication.