Implementing the tangent Graeffe
root finding method[*][†]

JORIS VAN DER HOEVEN[abc], MICHAEL MONAGAN[ad]

a. Department of Mathematics
Simon Fraser University
8888 University Drive
Burnaby, British Columbia
V5A 1S6, Canada

b. CNRS, École polytechnique, Institut Polytechnique de Paris
Laboratoire d'informatique de l'École polytechnique (LIX, UMR 7161)
1, rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing, CS35003
91120 Palaiseau, France

c. *Email:* `vdhoeven@lix.polytechnique.fr`
d. *Email:* `mmonagan@cecm.sfu.ca`

May 2, 2020

The tangent Graeffe method has been developed for the efficient computation of single roots of polynomials over finite fields with multiplicative groups of smooth order. It is a key ingredient of sparse interpolation using geometric progressions, in the case when blackbox evaluations are comparatively cheap. In this paper, we improve the complexity of the method by a constant factor and we report on a new implementation of the method and a first parallel implemenation.

---

1. INTRODUCTION

Consider a polynomial function $f: \mathbb{K}^n \to \mathbb{K}$ over a field $\mathbb{K}$ given through a black box capable of evaluating f at points in $\mathbb{K}^n$. The problem of *sparse interpolation* is to recover the representation of $f \in \mathbb{K}[x_1, \ldots, x_n]$ in its usual form, as a linear combination

$$f = \sum_{1 \leqslant i \leqslant t} c_i x^{e_i} \tag{1}$$

of monomials $x^{e_i} = x_1^{e_{1,1}} \cdots x_n^{e_{1,n}}$. One popular approach to sparse interpolation is to evaluate f at points in a geometric progression. This approach goes back to work of Prony in the eighteen's century [29] and became well known after Ben-Or and Tiwari's seminal paper [2]. It has widely been used in computer algebra, both in theory [6, 19, 21, 22, 23, 24, 28] and in practice [8, 9, 15, 16, 18, 20]; see [30] for a nice survey.

More precisely, if a bound T for the number of terms t is known, then we first evaluate f at $2T-1$ pairwise distinct points $\alpha^0, \alpha^1, \ldots, \alpha^{2T-2}$, where $\alpha = (\alpha_1, \ldots, \alpha_n) \in \mathbb{K}^n$ and $\alpha^k := (\alpha_1^k, \ldots, \alpha_n^k)$ for all $k \in \mathbb{N}$. The generating function of the evaluations at α^k satisfies the identity

$$\sum_{k \in \mathbb{N}} f(\alpha^k) z^k = \sum_{1 \leqslant i \leqslant t} \sum_{k \in \mathbb{N}} c_i \alpha^{e_i k} z^k = \sum_{1 \leqslant i \leqslant t} \frac{c_i}{1 - \alpha^{e_i} z} = \frac{N(z)}{\Lambda(z)},$$

---

where $\Lambda = (1 - \alpha^{e_1} z) \cdots (1 - \alpha^{e_t} z)$ and $N \in \mathbb{K}[z]$ is of degree $< t$. The rational function N/Λ can be recovered from $f(\alpha^0), f(\alpha^1), \ldots, f(\alpha^{2T-2})$ using fast Padé approximation [5, 27]. For well chosen points α, it is often possible to recover the exponents e_i from the values $\alpha^{e_i} \in \mathbb{K}$. If the exponents e_i are known, then the coefficients c_i can also be recovered using a transposed version of fast multipoint interpolation [4, 6]. This leaves us with the question how to compute the roots α^{-e_i} of Λ in an efficient way.

For practical applications in computer algebra, we usually have $\mathbb{K} = \mathbb{Q}$, in which case it is most efficient to use a multi-modular strategy. This means that we rather work with coefficients in a finite field $\mathbb{K} = \mathbb{F}_p$, where p is a prime number that we are free to choose. It is well known that polynomial arithmetic over $\mathbb{F}_p$ can be implemented most efficiently using FFTs when the order $p-1$ of the multiplicative group is smooth. In practice, this prompts us to choose p of the form $s\,2^l + 1$ for some small s and such that p fits into a machine word.

The traditional way to compute roots of polynomials over finite fields is using Cantor and Zassenhaus' method [7]. In [12, 13], alternative algorithms were proposed for our case of interest when $p-1$ is smooth. The fastest algorithm was based on the *tangent Graeffe transform* and it gains a factor $\log t$ with respect to Cantor–Zassenhaus' method. The aim of the present paper is to report on a parallel implementation of this new algorithm and on a few improvements that allow for a further constant speed-up.

In section 2, we start by recalling generalities about the Graeffe transform and the heuristic root finding method based on the tangent Graeffe transform from [12]. In section 3, we present the main new theoretical improvements, which all rely on optimizations in the FFT-model for fast polynomial arithmetic. Our contributions are threefold:

- In the FFT-model, one backward transform out of four can be saved for Graeffe transforms of order two (see section 3.2).

- When composing a large number of Graeffe transforms of order two, FFT caching can be used to gain another factor of $3/2$ (see section 3.3).

- All optimizations still apply in the TFT model, which can save a factor between one and two, depending on the number of roots (see section 3.5).

We also indicate how to generalize our methods to Graeffe transforms of general orders in section 3.4.

Section 4 is devoted to our new implementation. We first implemented a sequential version of the tangent Graeffe method in C, with the optimizations from sections 3.2 and 3.3. We also started experimenting with a parallel implementation in Cilk C. Our sequential implementation confirms the major performance improvement with respect to Cantor–Zassenhaus' algorithm. We also observed the gain of a new factor of two when using the new optimizations. So far, we have achieved a parallel speed-up by a factor of 4.6 on an 8-core machine. Our implementation is freely available at:

> http://www.cecm.sfu.ca/CAG/code/TangentGraeffe

2. ROOT FINDING USING THE TANGENT GRAEFFE TRANSFORM

2.1. Graeffe transforms

The traditional *Graeffe transform* of a monic polynomial $P \in \mathbb{K}[z]$ of degree d is the unique monic polynomial $G(P) \in \mathbb{K}[z]$ of degree d such that

$$G(P)(z^2) = P(z)\,P(-z). \tag{2}$$

If P splits over $\mathbb{K}$ into linear factors $P = (z - \beta_1) \cdots (z - \beta_d)$, then one has

$$G(P) = (z - \beta_1^2) \cdots (z - \beta_d^2).$$

More generally, given $r \geqslant 2$, we define the *Graeffe transform of order* r to be the unique monic polynomial $G_r(P) \in \mathbb{K}[z]$ of degree d such that

$$G_r(P)(z) = (-1)^{rd} \operatorname{Res}_u(P(u), u^r - z)$$

If $P = (z - \beta_1) \cdots (z - \beta_d)$, then

$$G_r(P) = (z - \beta_1^r) \cdots (z - \beta_d^r).$$

If ω is a primitive r-th root of unity in $\mathbb{K}$, then we also have

$$G_r(P)(z^r) = P(z) P(\omega z) \cdots P(\omega^{r-1} z). \tag{3}$$

If $r, s \geqslant 2$, then we finally have

$$G_{rs} = G_r \circ G_s = G_s \circ G_r. \tag{4}$$

2.2. Root finding using tangent Graeffe transforms

Let ϵ be a formal indeterminate with $\epsilon^2 = 0$. Elements in $\mathbb{K}[\epsilon]/(\epsilon^2)$ are called *tangent numbers*. They are of the form $a + b\epsilon$ with $a, b \in \mathbb{K}$ and basic arithmetic operations go as follows:

$$(a + b\epsilon) \pm (c + d\epsilon) = (a \pm c) + (b \pm d)\epsilon$$
$$(a + b\epsilon)(c + d\epsilon) = ac + (ad + bc)\epsilon$$

Now let $P \in \mathbb{K}[z]$ be a monic polynomial of degree d that splits over $\mathbb{K}$:

$$P = (z - \alpha_1) \cdots (z - \alpha_d),$$

where $\alpha_1, \ldots, \alpha_d \in \mathbb{K}$ are pairwise distinct. Then the *tangent deformation* $\tilde{P}(z) := P(z + \varepsilon)$ satisfies

$$\tilde{P} = P + P'\epsilon = (z - (\alpha_1 - \epsilon)) \cdots (z - (\alpha_d - \epsilon)).$$

The definitions from the previous subsection readily extend to coefficients in $\mathbb{K}[\epsilon]$ instead of $\mathbb{K}$. Given $r \geqslant 2$, we call $G_r(\tilde{P})$ the *tangent Graeffe transform* of P of order r. We have

$$G_r(\tilde{P}) = (z - (\alpha_1 - \epsilon)^r) \cdots (z - (\alpha_d - \epsilon)^r),$$

where

$$(\alpha_k - \epsilon)^r = \alpha_k^r - r\alpha_k^{r-1}\epsilon, \qquad k = 1, \ldots, d.$$

Now assume that we have an efficient way to determine the roots $\alpha_1^r, \ldots, \alpha_d^r$ of $G_r(P)$. For some polynomial $T \in \mathbb{K}[z]$, we may decompose

$$G_r(\tilde{P}) = G_r(P) + T\epsilon$$

For any root α_k^r of $G_r(P)$, we then have

$$\begin{aligned}
G_r(\tilde{P})(\alpha_k^r - r\alpha_k^{r-1}\epsilon) &= G_r(P)(\alpha_k^r) + (T(\alpha_k^r) - G_r(P)'(\alpha_k^r) r\alpha_k^{r-1})\epsilon \\
&= (T(\alpha_k^r) - G_r(P)'(\alpha_k^r) r\alpha_k^{r-1})\epsilon \\
&= 0.
\end{aligned}$$

Whenever α_k^r happens to be a single root of $G_r(P)$, it follows that

$$r\alpha_k^{r-1} = \frac{T(\alpha_k^r)}{G_r(P)'(\alpha_k^r)}.$$

If $\alpha_k^r \neq 0$, this finally allows us to recover α_k as

$$\alpha_k = r \frac{\alpha_k^r}{r\,\alpha_k^{r-1}}.$$

2.3. Heuristic root finding over smooth finite fields

Assume now that $\mathbb{K} = \mathbb{F}_p$ is a finite field, where p is a prime number of the form $p = \sigma 2^m + 1$ for some small σ. Assume also that $\omega \in \mathbb{F}_p$ be a primitive element of order $p - 1$ for the multiplicative group of $\mathbb{F}_p$.

Let $P = (z - \alpha_1) \cdots (z - \alpha_d) \in \mathbb{F}_p[z]$ be as in the previous subsection. The tangent Graeffe method can be used to efficiently compute those α_k of P for which α_k^r is a single root of $G_r(P)$. In order to guarantee that there are a sufficient number of such roots, we first replace $P(z)$ by $P(z + \tau)$ for a random shift $\tau \in \mathbb{F}_p$, and use the following heuristic:

> **H.** For any subset $\{\alpha_1, \ldots, \alpha_d\} \subseteq \mathbb{F}_p$ of cardinality d and any $r \leqslant (p-1)/(4d)$, there exist at least $p/2$ elements $\tau \in \mathbb{F}_p$ such that $\{(\alpha_1 - \tau)^r, \ldots, (\alpha_d - \tau)^r\}$ contains at least $2d/3$ elements.

For a random shift $\tau \in \mathbb{F}_p$ and any $r \leqslant (p-1)/(4d)$, the assumption ensures with probability at least $1/2$ that $G_r(P(z+\tau))$ has at least $d/3$ single roots.

Now take r to be the largest power of two such that $r \leqslant (p-1)/(4d)$ and let $s = (p-1)/r$. By construction, note that $s = O(d)$. The roots $\alpha_1^r, \ldots, \alpha_d^r$ of $G_r(P)$ are all s-th roots of unity in the set $\{1, \omega^r, \ldots, \omega^{(s-1)r}\}$. We may thus determine them by evaluating $G_r(P)$ at ω^i for $i = 0, \ldots, s-1$. Since $s = O(d)$, this can be done efficiently using a discrete Fourier transform. Combined with the tangent Graeffe method from the previous subsection, this leads to the following probabilistic algorithm for root finding:

Algorithm 1
  Input: $P \in \mathbb{F}_p[z]$ of degree d and only order one factors, $p = \sigma 2^m + 1$
Output: the set $\{\alpha_1, \ldots, \alpha_d\}$ of roots of P

---

1. If $d = 0$ then return $\emptyset$

2. Let $\tau \in \mathbb{F}_p$ be a random shift
   Let $r = 2^N \in 2^{\mathbb{N}}$ be largest such that $r \leqslant (p-1)/(4d)$ and let $s := (p-1)/r$

3. Compute $P^* := P(z + \tau) \in \mathbb{F}_p[z]$

4. Compute $\tilde{P}(z) := P^*(z + \epsilon) = P^*(z) + P^*(z)'\epsilon \in (\mathbb{F}_p[\epsilon]/(\epsilon^2))[z]$

5. For $i = 1, \ldots, N$, set $\tilde{P} := G_2(\tilde{P}) \in (\mathbb{F}_p[\epsilon]/(\epsilon^2))[z]$

6. Write $\tilde{P} = A + B\epsilon$ and compute $A(\omega^{ir})$, $A'(\omega^{ir})$, and $B(\omega^{ir})$ for $i = 0, \ldots, s-1$

7. If $P(\tau) = 0$, then set $S := \{\tau\}$, else set $S := \emptyset$

8. Let $\omega \in \mathbb{F}_p^*$ be of order $p - 1$
   For any $\beta \in \{1, \omega^r, \ldots, \omega^{(s-1)r}\}$ with $A(\beta) = 0$ and $A'(\beta) \neq 0$,
   let $S := S \cup \{r\beta A'(\beta)/B(\beta) + \tau\}$

9. Compute $Q := \prod_{\alpha \in S} (z - \alpha)$

10. Recursively determine the set of roots S' of P/Q

11. Return $S \cup S'$

Remark 1. To compute $G_2(\tilde{P}) = G_2(A + B\,\epsilon)$ we may use $G_2(\tilde{P}(z^2)) = A(z)\,A(-z) + (A(z)\,B(-z) + B(z)\,A(-z))\,\epsilon$, which requires three polynomial multiplications in $\mathbb{F}_p[z]$ of degree d. In total, step 5 therefore performs $O(N) = O(\log(p/s))$ such multiplications. We discuss how to perform step 5 efficiently in the FFT model in section 3.

Remark 2. For practical implementations, one may vary the threshold $r \leqslant (p-1)/(4d)$ for r and the resulting threshold $s \geqslant 4d$ for s. For larger values of s, the computations of the DFTs in step 6 get more expensive, but the proportion of single roots goes up, so more roots are determined at each iteration. From an asymptotic complexity perspective, it would be best to take $s \asymp d\sqrt{\log p}$. In practice, we actually preferred to take the *lower* threshold $s \geqslant 2d$, because the constant factor of our implementation of step 6 (based on Bluestein's algorithm [3]) is significant with respect to our highly optimized implementation of the tangent Graeffe method. A second reason we prefer s of size $O(d)$ instead of $O(d\sqrt{\log p})$ is that the total space used by the algorithm is linear in s. In the future, it would be interesting to further speed up step 6 by investing more time in the implementation of high performance DFTs of general orders s.

Remark 3. For the application to sparse interpolation, it is possible to further speed up step 5 for the top-level iteration, which is the most expensive step. More precisely, for a polynomial with t terms, the idea is to take $\tau = 0$ and ω of order $\approx t^c$ instead of $p-1$ for some constant c with $1 < c < 3$. This reduces $\log r$ (and the cost of the top-level iteration) by a factor of $\Theta(\log p/\log t)$. For the recursive calls, we still need to work with a primitive root of unity ω' of order $p-1$ and random shifts.

3. COMPUTING GRAEFFE TRANSFORMS

3.1. Reminders about discrete Fourier transforms

Assume that $n \in \mathbb{N}$ is invertible in $\mathbb{K}$ and let $\omega \in \mathbb{K}$ be a primitive n-th root of unity. Consider a polynomial $A = a_0 + a_1 z + \cdots + a_{n-1} z^{n-1} \in \mathbb{K}[z]$. Then the discrete Fourier transform (DFT) of order n of the sequence $(a_i)_{0 \leqslant i < n}$ is defined by

$$\mathrm{DFT}_\omega((a_i)_{0 \leqslant i < n}) := (\hat{a}_k)_{0 \leqslant k < n}, \qquad \hat{a}_k := A(\omega^k).$$

We will write $\mathsf{F}_\mathbb{K}(n)$ for the cost of one discrete Fourier transform in terms of the number of operations in $\mathbb{K}$ and assume that $n = o(\mathsf{F}_\mathbb{K}(n))$. For any $i \in \{0, \ldots, n-1\}$, we have

$$\mathrm{DFT}_{\omega^{-1}}((\hat{a}_k)_{0 \leqslant k < n})_i = \sum_{0 \leqslant k < n} \hat{a}_k \omega^{-ik} = \sum_{0 \leqslant j < n} a_j \sum_{0 \leqslant k < n} \omega^{(j-i)k} = n\,a_i. \tag{5}$$

If n is invertible in $\mathbb{K}$, then it follows that $\mathrm{DFT}_\omega^{-1} = n^{-1}\,\mathrm{DFT}_{\omega^{-1}}$. The costs of direct and inverse transforms therefore coincide up to a factor $O(n)$.

If $n = n_1 n_2$ is composite, $0 \leqslant k_1 < n_1$, and $0 \leqslant k_2 < n_2$, then we have

$$\hat{a}_{k_2 n_1 + k_1} = \sum_{0 \leqslant i_2 < n_2} \sum_{0 \leqslant i_1 < n_1} a_{i_1 n_2 + i_2} \omega^{(i_1 n_2 + i_2)(k_2 n_1 + k_1)}$$

$$= \sum_{0 \leqslant i_2 < n_2} \omega^{i_2 k_1} \left[\sum_{0 \leqslant i_1 < n_1} a_{i_1 n_2 + i_2} \omega^{i_1 n_2 k_1} \right] \omega^{i_2 k_2 n_1}$$

$$= \sum_{0 \leqslant i_2 < n_2} [\omega^{i_2 k_1} \mathrm{DFT}_{\omega^{n_2}}((a_{i_1 n_2 + i_2})_{0 \leqslant i_1 < n_1})_{k_1}] \omega^{i_2(k_2 n_1 + k_1)}$$

$$= \mathrm{DFT}_{\omega^{n_1}}((\omega^{i_2 k_1} \mathrm{DFT}_{\omega^{n_2}}((a_{i_1 n_2 + i_2})_{0 \leqslant i_1 < n_1})_{k_1})_{0 \leqslant i_2 < n_2})_{k_2}. \tag{6}$$

This shows that a DFT of length n reduces to n_1 transforms of length n_2 plus n_2 transforms of length n_1 plus n multiplications in $\mathbb{K}$:

$$\mathsf{F}_{\mathbb{K}}(n_1 n_2) \leqslant n_1 \mathsf{F}_{\mathbb{K}}(n_2) + n_2 \mathsf{F}_{\mathbb{K}}(n_1) + O(n).$$

In particular, if $r = O(1)$, then $\mathsf{F}_{\mathbb{K}}(r n) \sim r \mathsf{F}_{\mathbb{K}}(n)$.

It is sometimes convenient to apply DFTs directly to polynomials as well; for this reason, we also define $\mathrm{DFT}_\omega(A) := (\hat{a}_k)_{0 \leqslant k < n}$. Given two polynomials $A, B \in \mathbb{K}[z]$ with $\deg(A B) < n$, we may then compute the product $A B$ using

$$A B \;=\; \mathrm{DFT}_\omega^{-1}(\mathrm{DFT}_\omega(A)\, \mathrm{DFT}_\omega(B)).$$

In particular, if $\mathsf{M}_{\mathbb{K}}(n)$ denotes the cost of multiplying two polynomials of degree $<n$, then we obtain $\mathsf{M}_{\mathbb{K}}(n) \sim 3\, \mathsf{F}_{\mathbb{K}}(2n) \sim 6\, \mathsf{F}_{\mathbb{K}}(n)$.

Remark 4. In Algorithm 1, we note that step 6 comes down to the computation of three DFTs of length s. Since r is a power of two, this length is of the form $s = \sigma\, 2^k$ for some $k \in \mathbb{N}$. In view of (6), we may therefore reduce step 6 to 3σ DFTs of length 2^k plus $3 \cdot 2^k$ DFTs of length σ. If σ is very small, then we may use a naive implementation for DFTs of length σ. In general, one may use Bluestein's algorithm [3] to reduce the computation of a DFT of length σ into the computation of a product in $\mathbb{K}[z]/(z^\sigma - 1)$, which can in turn be computed using FFT-multiplication and three DFTs of length a larger power of two.

3.2. Graeffe transforms of order two

Let $\mathbb{K}$ be a field with a primitive $(2n)$-th root of unity ω. Let $P \in \mathbb{K}[z]$ be a polynomial of degree $d = \deg P < n$. Then the relation (2) yields

$$G(P)(z^2) = \mathrm{DFT}_\omega^{-1}(\mathrm{DFT}_\omega(P(z))\, \mathrm{DFT}_\omega(P(-z))). \tag{7}$$

For any $k \in \{0, \dots, 2n-1\}$, we further note that

$$\mathrm{DFT}_\omega(P(-z))_k = P(-\omega^k) = P(\omega^{(k+n)\,\mathrm{rem}\,2n}) = \mathrm{DFT}_\omega(P(z))_k, \tag{8}$$

so $\mathrm{DFT}_\omega(P(-z))$ can be obtained from $\mathrm{DFT}_\omega(P)$ using n transpositions of elements in $\mathbb{K}$. Concerning the inverse transform, we also note that

$$\mathrm{DFT}_\omega(G(P)(z^2))_k = G(P)(\omega^{2k}) = \mathrm{DFT}_{\omega^2}(G(P))_k,$$

for $k = 0, \dots, n-1$. Plugging this into (7), we conclude that

$$G(P) = \mathrm{DFT}_{\omega^2}^{-1}((\mathrm{DFT}_\omega(P)_{2k}\, \mathrm{DFT}_\omega(P)_{(2k+n)\,\mathrm{rem}\,2n})_{0 \leqslant k < n}).$$

This leads to the following algorithm for the computation of $G(P)$:

Algorithm 2

Input: $P \in \mathbb{K}[z]$ with $\deg P < n$ and a primitive $(2n)$-th root of unity $\omega \in \mathbb{K}$
Output: $G(P)$

---

1. Compute $(\hat{P}_{0,k})_{0 \leqslant k < 2n} := \mathrm{DFT}_\omega(P)$

2. For $k = 0, \dots, n-1$, set $\hat{P}_{1,2k} := \hat{P}_{0,(2k+n)\,\mathrm{rem}\,2n}$

3. For $k = 0, \dots, n-1$, compute $\hat{G}_{2k} := \hat{P}_{0,2k}\, \hat{P}_{1,2k}$

4. Return $\mathrm{DFT}_{\omega^2}^{-1}((\hat{G}_{2k})_{0 \leqslant k < n})$

PROPOSITION 5. *Let $\omega \in \mathbb{K}$ be a primitive $2n$-th root of unity in $\mathbb{K}$ and assume that 2 is invertible in $\mathbb{K}$. Given a monic polynomial $P \in \mathbb{K}[z]$ with $\deg P < n$, we can compute $G(P)$ in time*

$$\mathsf{G}_{2,\mathbb{K}}(n) \sim 3\,\mathsf{F}_{\mathbb{K}}(n).$$

Proof. We have already explained the correctness of Algorithm 2. Step 1 requires one forward DFT of length $2n$ and cost $\mathsf{F}_{\mathbb{K}}(2n) = 2\,\mathsf{F}_{\mathbb{K}}(n) + O(n)$. Steps 2 and 3 can be done in linear time $O(n)$. Step 4 requires one inverse DFT of length n and cost $\mathsf{F}_{\mathbb{K}}(n) + O(n)$. The total cost of Algorithm 2 is therefore $3\,\mathsf{F}_{\mathbb{K}}(n) + O(n) \sim 3\,\mathsf{F}_{\mathbb{K}}(n)$.   □

Remark 6. In terms of the complexity of multiplication, we obtain $\mathsf{G}_{2,\mathbb{K}}(n) \sim (1/2)\,\mathsf{M}_{\mathbb{K}}(n)$. This gives a 33.3% improvement over the previously best known bound $\mathsf{G}_{2,\mathbb{K}}(n) \sim (2/3)\,\mathsf{M}_{\mathbb{K}}(n)$ that was used in [12]. Note that the best known algorithm for computing squares of polynomials of degree $<n$ is $\sim (2/3)\,\mathsf{M}_{\mathbb{K}}(n)$. It would be interesting to know whether squares can also be computed in time $\sim (1/2)\,\mathsf{M}_{\mathbb{K}}(n)$.

3.3. Graeffe transforms of power of two orders

In view of (4), Graeffe transforms of power of two orders 2^m can be computed using

$$G_{2^m}(P) = (G \circ \overset{m\times}{\cdots} \circ G)(P). \tag{9}$$

Now assume that we computed the first Graeffe transform $G(P)$ using Algorithm 2 and that we wish to apply a second Graeffe transform to the result. Then we note that

$$\mathrm{DFT}_\omega(G(P))_{2k} = \mathrm{DFT}_{\omega^2}(G(P))_k = \hat{G}_{2k} \tag{10}$$

is already known for $k = 0, \ldots, n-1$. We can use this to accelerate step 1 of the second application of Algorithm 2. Indeed, in view of (6) for $n_1 = 2$ and $n_2 = n$, we have

$$\mathrm{DFT}_\omega(G(P))_{2k+1} = \mathrm{DFT}_{\omega^2}((\omega^i G(P)_i)_{0 \leqslant i < n})_k \tag{11}$$

for $k = 0, \ldots, n-1$. In order to exploit this idea in a recursive fashion, it is useful to modify Algorithm 2 so as to include $\mathrm{DFT}_{\omega^2}(P)$ in the input and $\mathrm{DFT}_{\omega^2}(G(P))$ in the output. This leads to the following algorithm:

Algorithm 3
  Input: $P \in \mathbb{K}[z]$ with $\deg P < n$, a primitive $(2n)$-th root of unity $\omega \in \mathbb{K}$,
       and $(\hat{P}_k)_{0 \leqslant k < n} = \mathrm{DFT}_{\omega^2}(P)$
Output: $G(P)$ and $\mathrm{DFT}_{\omega^2}(G(P))$

---
1. Set $(\hat{P}_{0,2k})_{0 \leqslant k < n} := (\hat{P}_k)_{0 \leqslant k < n}$

2. Set $(\hat{P}_{0,2k+1})_{0 \leqslant k < n} := \mathrm{DFT}_{\omega^2}((\omega^i P_i)_{0 \leqslant i < n})$

3. For $k = 0, \ldots, n-1$, set $\hat{P}_{1,2k} := \hat{P}_{0,(2k+n)\,\mathrm{rem}\,2n}$

4. For $k = 0, \ldots, n-1$, compute $\hat{G}_{2k} := \hat{P}_{0,2k} \hat{P}_{1,2k}$

5. Return $\mathrm{DFT}_{\omega^2}^{-1}((\hat{G}_{2k})_{0 \leqslant k < n})$ and $(\hat{G}_{2k})_{0 \leqslant k < n}$

PROPOSITION 7. *Let $\omega \in \mathbb{K}$ be a primitive $2n$-th root of unity in $\mathbb{K}$ and assume that 2 is invertible in $\mathbb{K}$. Given a monic polynomial $P \in \mathbb{K}[z]$ with $\deg P < n$ and $m \geqslant 1$, we can compute $G_{2^m}(P)$ in time*

$$\mathsf{G}_{2^m,\mathbb{K}}(n) \sim (2m+1)\,\mathsf{F}_{\mathbb{K}}(n).$$

Proof. It suffices to compute $\mathrm{DFT}_{\omega^2}(P)$ and then to apply Algorithm 3 recursively, m times. Every application of Algorithm 3 now takes $2\,\mathsf{F}_\mathbb{K}(n) + O(n) \sim 2\,\mathsf{F}_\mathbb{K}(n)$ operations in $\mathbb{K}$, whence the claimed complexity bound. □

Remark 8. In [12], Graeffe transforms of order 2^m were directly computed using the formula (9), using $\sim 4\,m\,\mathsf{F}_\mathbb{K}(n)$ operations in $\mathbb{K}$. The new algorithm is twice as fast for large m.

3.4. Graeffe transforms of arbitrary smooth orders

The algorithms from subsections 3.2 and 3.3 readily generalize to Graeffe transforms of order r^m for arbitrary $r \geqslant 2$, provided that we have an $(r\,n)$-th root of unity $\omega \in \mathbb{K}$. For convenience of the reader, we specified the generalization of Algorithm 3 below, together with the resulting complexity bounds.

Algorithm 4

  Input: $P \in \mathbb{K}[z]$ with $\deg P < n$, $r \geqslant 2$, a primitive $(r\,n)$-th root of unity $\omega \in \mathbb{K}$,
        and $(\hat{P}_k)_{0 \leqslant k < n} = \mathrm{DFT}_{\omega^r}(P)$

Output: $G_r(P)$ and $\mathrm{DFT}_{\omega^r}(G_r(P))$

1. Set $(\hat{P}_{0,kr})_{0 \leqslant k < n} := (\hat{P}_k)_{0 \leqslant k < n}$

2. For $j = 1, \ldots, r-1$, set $(\hat{P}_{0,kr+j})_{0 \leqslant k < n} := \mathrm{DFT}_{\omega^r}((\omega^{ij} P_i)_{0 \leqslant i < n})$

3. For $j = 1, \ldots, r-1$ and $k = 0, \ldots, n-1$, set $\hat{P}_{j,kr} := \hat{P}_{0,(kr+jn)\,\mathrm{rem}\,rn}$

4. For $k = 0, \ldots, n-1$, compute $\hat{G}_{kr} := \hat{P}_{0,kr} \hat{P}_{1,kr} \cdots \hat{P}_{r-1,kr}$

5. Return $\mathrm{DFT}_{\omega^r}^{-1}((\hat{G}_{kr})_{0 \leqslant k < n})$ and $(\hat{G}_{kr})_{0 \leqslant k < n}$

PROPOSITION 9. *Let $\omega \in \mathbb{K}$ be a primitive $(r\,n)$-th root of unity in $\mathbb{K}$, where $r \geqslant 2$ is invertible in $\mathbb{K}$. Given a monic polynomial $P \in \mathbb{K}[z]$ with $\deg P < n$ and $m \geqslant 1$, we can compute $G_{r^m}(P)$ in time*

$$\mathsf{G}_{r^m,\mathbb{K}}(n) \sim (r\,m+1)\,\mathsf{F}_\mathbb{K}(n).$$

Proof. Straightforward generalization of Proposition 7. □

COROLLARY 10. *Let $\omega \in \mathbb{K}$ be a primitive $(r_1 \cdots r_\tau n)$-th root of unity in $\mathbb{K}$, where $r_1 \geqslant 2, \ldots,$ $r_\tau \geqslant 2$ are invertible in $\mathbb{K}$. Given a monic polynomial $P \in \mathbb{K}[z]$ with $\deg P < n$ and $m_1,...,m_\tau \in \mathbb{N}$, we can compute $G_{r_1^{m_1} \cdots r_\tau^{m_\tau}}(P)$ in time*

$$\mathsf{G}_{r_1^{m_1} \cdots r_\tau^{m_\tau},\mathbb{K}}(n) \sim (r_1 m_1 + \cdots + r_\tau m_\tau + \tau)\,\mathsf{F}_\mathbb{K}(n).$$

Proof. Direct consequence of (4). □

Remark 11. In our application to root finding, we are interested in the efficient computation of Graeffe transforms of high order r^m. In terms of the size $\log r^m$ of r^m, it is instructive to observe that the "average cost"

$$\mathsf{A}_{r^m}(n) = \frac{\mathsf{G}_{r^m,\mathbb{K}}(n)}{\log r^m\,\mathsf{F}_\mathbb{K}(n)} \sim \frac{r}{\log r}$$

is minimal for $r = 3$. Whenever radix-3 DFTs can be implemented as efficiently as radix-2 DFTs, then it follows that it might be interesting to use Graeffe transforms of order three.

3.5. Truncated Fourier transforms

If $\mathbb{K} = \mathbb{F}_q$ is a fixed finite field, then DFTs are most efficient for sizes n that divide $q - 1$. For our root finding application, it is often convenient to take $q = 3 \cdot 2^{30} + 1$, in which case n should be a power of two or three times a power of two. The truncated Fourier transform was developed for the multiplication of polynomials such that the degree of the product does not have a nice size n of this type. It turns out that we may also use it for the efficient computation of Graeffe transforms of polynomials of arbitrary degrees. Moreover, the optimizations from the previous subsections still apply.

Let us briefly describe how the truncated Fourier transform can be used for the computation of Graeffe transforms of power of two orders. With the notations from subsections 3.2 and 3.3, we assume that $2n = 2^\beta$ is a power of two as well and that we wish to compute the Graeffe transform of a polynomial P of degree $\deg P < t$ with $n/2 \leqslant t < n$. Let $[i]_\beta$ denote the reversal of a binary number $i \in \{0, \dots, 2n - 1\}$ of β bits. For instance, $[3]_4 = 12$ and $[5]_6 = 40$. Then the truncated Fourier of P at order $T \geqslant t$ is defined by

$$\mathrm{TFT}_{\omega,T}(P) = (P(\omega^{[0]_\beta}), P(\omega^{[1]_\beta}), \dots, P(\omega^{[T-1]_\beta})).$$

It has been shown in [14] that $\tilde{P} := \mathrm{TFT}_{\omega,T}(P)$ and $P = \mathrm{TFT}_{\omega,T}^{-1}(\tilde{P})$ can both be computed in time $\sim (T/n)\, \mathsf{F}_\mathbb{K}(n)$. For generalizations to arbitrary radices, we refer to [25].

Taking $T = 2t$, we notice that

$$P(\omega^{[2k+1]_\beta}) = P(\omega^{[2k]_\beta + n/2}) = P(-\omega^{[2k]_\beta})$$

for $k = 0, \dots, t - 1$. This provides us with the required counterpart of (8) for retrieving $\mathrm{TFT}_{\omega,2t}(P(-x))$ efficiently from $\mathrm{TFT}_{\omega,2t}(P)$. The relation (10) also has a natural counterpart:

$$\mathrm{TFT}_{\omega,2t}(G(P))_{2k} = \mathrm{TFT}_{\omega^2,t}(G(P))_k,$$

for $k = 0, \dots, t - 1$, and so does (11):

$$\mathrm{TFT}_{\omega,2t}(G(P))_{2k+1} = \mathrm{DFT}_{\omega^2,t}(G(P)(\omega z))_k.$$

This leads to the following refinement of Algorithm 3:

Algorithm 5

Input: $P \in \mathbb{K}[z]$ with $\deg P < t \leqslant n = 2^{\beta-1}$,
  a primitive $(2n)$-th root of unity $\omega \in \mathbb{K}$, and $(\hat{P}_k)_{0 \leqslant k < t} = \mathrm{TFT}_{\omega^2,t}(P)$
Output: $G(P)$ and $\mathrm{TFT}_{\omega^2,t}(G(P))$

---

1. Set $(\hat{P}_{0,2k})_{0 \leqslant k < t} := (\hat{P}_k)_{0 \leqslant k < t}$

2. Set $(\hat{P}_{0,2k+1})_{0 \leqslant k < t} := \mathrm{TFT}_{\omega^2,t}(P(\omega z))$

3. For $k = 0, \dots, t - 1$, set $\hat{P}_{1,2k} := \hat{P}_{0,2k+1}$

4. For $k = 0, \dots, t - 1$, compute $\hat{G}_{2k} := \hat{P}_{0,2k} \hat{P}_{1,2k}$

5. Return $\mathrm{TFT}_{\omega^2,t}^{-1}((\hat{G}_{2k})_{0 \leqslant k < t})$ and $(\hat{G}_{2k})_{0 \leqslant k < t}$

PROPOSITION 12. *Let $\omega \in \mathbb{K}$ be a primitive $2n$-th root of unity in $\mathbb{K}$, where $2n = 2^{\beta}$, and assume that 2 is invertible in $\mathbb{K}$. Given a monic polynomial $P \in \mathbb{K}[z]$ with $n/2 \leqslant \deg P < t \leqslant n$ and $m \geqslant 1$, we can compute $G_{2^m}(P)$ in time*

$$G_{2^m, \mathbb{K}}(t;n) \sim \frac{t}{n}(2m+1)\, \mathsf{F}_{\mathbb{K}}(n).$$

Proof. Straightforward adaptation of the proof of Proposition 7, while using [14]. $\qquad\square$

3.6. Taylor shifts

In step 3 of Algorithm 1, we still need an algorithm to compute the Taylor shift $P(z+\tau)$. If the characteristic of $\mathbb{K}$ exceeds d, then it is (not so) well known [1, Lemma 3] that this can be reduced to a single polynomial multiplication of degree d using the following algorithm:

Algorithm 6
  Input: $P \in \mathbb{K}[z]$ of degree $d < \operatorname{char} \mathbb{K}$ and $\tau \in \mathbb{K}$
Output: $P(z+\tau)$

---

1. $L := 0!\,P_0 + 1!\,P_1 z + \cdots + d!\,P_d z^d$

2. $\tilde{L} := z^d L(1/z)$

3. $E := 1 + \tau z + \frac{1}{2!}\tau^2 z^2 + \cdots + \frac{1}{d!}\tau^d z^d$

4. $\tilde{\Pi} := \tilde{L} E \operatorname{rem} z^{d+1}$

5. $\Pi := z^d \tilde{\Pi}(1/z)$

6. Return $\frac{1}{0!}\Pi_0 + \frac{1}{1!}\Pi_1 z + \cdots + \frac{1}{d!}\Pi_d z^d$

It is interesting to observe that Taylor shifts can still be computed in time $O(\mathsf{M}(d))$ in small characteristic, as follows:

Algorithm 7
  Input: $P \in \mathbb{K}[z]$ of degree $d \geqslant p = \operatorname{char} \mathbb{K} > 0$ and $\tau \in \mathbb{K}$
Output: $P(z+\tau)$

---

1. Define $z_i = z^{p^i}$ for $i = 0, \ldots, k$ where $k = \lfloor \log d / \log p \rfloor$

2. Rewrite $P = \hat{P}(z_0, \ldots, z_k) \in \mathbb{K}[z_0, \ldots, z_k]$ with $\deg_{z_i} P < p$ for $i = 0, \ldots, k-1$

3. For $i = 0, \ldots, k$, replace $\hat{P} := \hat{P}\big(z_0, \ldots, z_{i-1}, z_i + \tau^{p^i}, z_{i+1}, \ldots, z_k\big)$

4. Return $\hat{P}\big(z, z^p, \ldots, z^{p^k}\big)$

---

4. IMPLEMENTATION AND BENCHMARKS

We have implemented the tangent Graeffe root finding algorithm (Algorithm 1) in C with the optimizations presented in section 3. Our C implementation supports primes of size up to 63 bits. In what follows all complexities count arithmetic operations in $\mathbb{F}_p$.

In Tables 1 and 2, the input polynomial $P(z)$ of degree d is constructed by choosing d distinct values $\alpha_i \in \mathbb{F}_p$ for $1 \leqslant i \leqslant d$ at random and creating $P(z) = \prod_{i=1}^{d}(z-\alpha_i)$. We will use $p = 3 \times 29 \times 2^{56} + 1$, a smooth 63 bit prime. For this prime $\mathsf{M}(d)$ is $O(d \log d)$.

One goal we have is to determine how much faster the Tangent Graeffe (TG) root finding algorithm is in practice when compared with the Cantor-Zassenhaus (CZ) algorithm which is implemented in many computer algebra systems. In Table 1 we present

		Our serial TG implementation in C					Magma CZ timings	
d	total	first	%roots	step 5	step 6	step 9	V2.25-3	V2.25-5
$2^{12}-1$	0.11s	0.07s	69.8%	0.04s	0.02s	0.01s	23.22s	8.43
$2^{13}-1$	0.22s	0.14s	69.8%	0.09s	0.03s	0.01s	56.58s	18.94
$2^{14}-1$	0.48s	0.31s	68.8%	0.18s	0.07s	0.02s	140.76s	44.07
$2^{15}-1$	1.00s	0.64s	69.2%	0.38s	0.16s	0.04s	372.22s	103.5
$2^{16}-1$	2.11s	1.36s	68.9%	0.78s	0.35s	0.10s	1494.0s	234.2
$2^{17}-1$	4.40s	2.85s	69.2%	1.62s	0.74s	0.23s	6108.8s	534.5
$2^{18}-1$	9.16s	5.91s	69.2%	3.33s	1.53s	0.51s	NA	1219.
$2^{19}-1$	19.2s	12.4s	69.2%	6.86s	3.25s	1.13s	NA	2809.

Table 1. Sequential timings in CPU seconds ($p = 3 \cdot 29 \cdot 2^{56} + 1$)

timings comparing our sequential implementation of the TG algorithm with Magma's implementation of the CZ algorithm. For polynomials in $\mathbb{F}_p[z]$, Magma uses Shoup's factorization algorithm from [31]. For our input $P(z)$, with d distinct linear factors, Shoup uses the Cantor–Zassenhaus equal degree factorization method. Recall that the average complexity of TG is $O(\mathsf{M}(d) (\log (p/s) + \log d))$ and of CZ is $O(\mathsf{M}(d) \log p \log d)$. We also wanted to attempt a parallel implementation. To do this we used the MIT Cilk C compiler from [10]. The Cilk C compiler provides a simple fork-join model of parallelism. Unlike the CZ algo- rithm, TG has no gcd computations that are hard to parallelize. We present some initial parallel timing data in Table 2.

The timings in Table 1 are sequential timings obtained on a a Linux server with an Intel Xeon E5-2660 CPU with 8 cores. In Table 1 the time in column "first" is for the first application of the TG algorithm (steps 1–9 of Algorithm 1) showing that it obtains about 69% of the roots. The time in column "total" is the total time for the algorithm. Columns step 5, step 6, and step 9 report the time spent in steps 5, 6, and 9 in Algorithm 1 and do not count time in the recursive call in step 10.

The Magma column timing is for Magma's `Factorization` command. The timings for Magma version V2.25-3 suggest that Magma's CZ implementation involves a subal-gorithm with quadratic asymptotic complexity. Indeed it turns out that the author of the code implemented all of the sub-quadratic polynomial arithmetic correctly, as demon-strated by the second set of timings for Magma in column V2.25-5, but inserted the d linear factors found into a list using linear insertion! Allan Steel of the Magma group identified and fixed the offending subroutine for Magma version V2.25-5. The timings show that TG is faster than CZ by a factor of 76.6 (=8.43/0.11) to 146.3 (=2809/19.2).

	Our parallel tangent Graeffe implementation in Cilk C				
d	total	first	step 5	step 6	step 9
$2^{19}-1$	18.30s(9.616s)	11.98s(2.938s)	6.64s(1.56s)	3.13s(0.49s)	1.09s(0.29s)
$2^{20}-1$	38.69s(12.40s)	25.02s(5.638s)	13.7s(3.03s)	6.62s(1.04s)	2.40s(0.36s)
$2^{21}-1$	79.63s(20.16s)	52.00s(11.52s)	28.1s(5.99s)	13.9s(2.15s)	5.32s(0.85s)
$2^{22}-1$	166.9s(41.62s)	107.8s(23.25s)	57.6s(11.8s)	28.9s(4.57s)	11.7s(1.71s)
$2^{23}-1$	346.0s(76.64s)	223.4s(46.94s)	117.s(23.2s)	60.3s(9.45s)	25.6s(3.54s)
$2^{24}-1$	712.7s(155.0s)	459.8s(95.93s)	238.s(46.7s)	125.s(19.17)	55.8s(7.88s)
$2^{25}-1$	1465.s(307.7s)	945.0s(194.6s)	481.s(92.9s)	259.s(39.2s)	121.s(16.9s)

Table 2. Real times in seconds for 1 core (8 cores) and $p = 3 \cdot 29 \cdot 2^{56} + 1$

4.1. Implementation notes

To implement the Taylor shift $P(z + \tau)$ in step 3, we used the $O(\mathsf{M}(d))$ method from [1, Lemma 3]. The better known method presented in [11] is $O(\mathsf{M}(d) \log d)$. For step 5 we use Algorithm 3 as presented. It has complexity $O(\mathsf{M}(d) \log p)$. To evaluate $A(z), A'(z)$ and $B(z)$ in step 6 in $O(\mathsf{M}(s))$ we used the Bluestein transformation [3]. In step 9 to compute the product $Q(z) = \Pi_{\alpha \in S} (z - \alpha)$, for $t = |S|$ roots, we used the $O(\mathsf{M}(t) \log t)$ tree multiplication algorithm [11]. The division in step 10 is done in $O(\mathsf{M}(d))$ with the fast division.

The sequential timings in Tables 1 and 2 show that steps 5, 6 and 9 account for about 91% of the total time. We parallelized these three steps as follows. For step 5, the two forward and two inverse FFTs are done in parallel. We also parallelized our radix 2 FFT by parallelizing recursive calls for size $n \geqslant 2^{17}$ and the main loop in blocks of size $m \geqslant 2^{18}$ as done in [26]. For step 6 there are three applications of Bluestein to compute $A(\omega^{ir})$, $A'(\omega^{ir})$ and $B(\omega^{ir})$. We parallelized these (thereby doubling the overall space used by our implementation). The main computation in the Bluestein transformation is a polynomial multiplication of two polynomials of degree s. The two forward FFTs are done in parallel and the FFTs themselves are parallelized as for step 5. For the product in step 9 we parallelize the two recursive calls in the tree multiplication for large sizes and again, the FFTs are parallelized as for step 5.

To improve parallel speedup we also parallelized the polynomial multiplication in step 3 and the computation of the roots in step 8. Although step 8 is $O(|S|)$, it is relatively expensive because of two inverse computations in $\mathbb{F}_p$. Because we have not parallelized about 5% of the computation the maximum parallel speedup we can obtain is a factor of $1 / (0.05 + 0.95/8) = 5.9$. The best overall parallel speedup we obtained is a factor of 4.6=1465/307.7 for $d = 2^{25} - 1$.

In Cilk, for each recursive C subroutine we wish to parallelize, one first needs to make a parallel version of that routine. For large recursive calls, the parallel version does them in parallel. For smaller recursive calls, the parallel version needs to call the sequential version of the code to avoid the Cilk process management overhead. The cutoff for the size of the input for which we need to use the sequential code has to be determined by experiment.

BIBLIOGRAPHY

[1]   A. V. Aho, K. Steiglitz, and J. D. Ullman. Evaluating polynomials on a fixed set of points. *SIAM Journ. of Comp.*, 4:533–539, 1975.

[2]   M. Ben-Or and P. Tiwari. A deterministic algorithm for sparse multivariate polynomial interpolation. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 301–309. ACM Press, 1988.

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

[4]   A. Bostan, G. Lecerf, and É. Schost. Tellegen's principle into practice. In *ISSAC '03: Proceedings of the 2003 International Symposium on Symbolic and Algebraic Computation*, pages 37–44. ACM Press, 2003.

[5]   R. P. Brent, F. G. Gustavson, and D. Y. Y. Yun. Fast solution of Toeplitz systems of equations and computation of Padé approximants. *J. Algorithms*, 1(3):259–295, 1980.

[6]   J. Canny, E. Kaltofen, and Y. Lakshman. Solving systems of non-linear polynomial equations faster. In *Proceedings of the ACM-SIGSAM 1989 International Symposium on Symbolic and Algebraic Computation*, pages 121–128. ACM Press, 1989.

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

[8] A. Díaz and E. Kaltofen. FOXFOX: a system for manipulating symbolic objects in black box representation. In *ISSAC '98: Proceedings of the 1998 International Symposium on Symbolic and Algebraic Computation*, pages 30–37. ACM Press, 1998.

[9] T. S. Freeman, G. M. Imirzian, E. Kaltofen, and Y. Lakshman. DAGWOOD: a system for manipulating polynomials given by straight-line programs. *ACM Trans. Math. Software*, 14:218–240, 1988.

[10] M. Frigo, C.E. Leisorson, and R.K. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of PLDI 1998*, pages 212–223. ACM, 1998.

[11] J. von zur Gathen and J. Gerhard. *Modern computer algebra*. Cambridge University Press, New York, 3rd edition, 2013.

[12] B. Grenet, J. van der Hoeven, and G. Lecerf. Randomized root finding over finite fields using tangent Graeffe transforms. In *Proc. ISSAC '15*, pages 197–204. New York, NY, USA, 2015. ACM.

[13] B. Grenet, J. van der Hoeven, and G. Lecerf. Deterministic root finding over finite fields using Graeffe transforms. *AAECC*, 27(3):237–257, 2016.

[14] J. van der Hoeven. The truncated Fourier transform and applications. In J. Gutierrez, editor, *Proc. ISSAC 2004*, pages 290–296. Univ. of Cantabria, Santander, Spain, July 4–7 2004.

[15] J. van der Hoeven and G. Lecerf. On the bit-complexity of sparse polynomial and series multiplication. *J. Symbolic Comput.*, 50:227–254, 2013.

[16] J. van der Hoeven and G. Lecerf. Sparse polynomial interpolation in practice. *ACM Commun. Comput. Algebra*, 48(3/4):187–191, 2015.

[17] J. van der Hoeven et al. GNU TeXmacs. http://www.texmacs.org, 1998.

[18] J. Hu and M. B. Monagan. A fast parallel sparse polynomial GCD algorithm. In S. A. Abramov, E. V. Zima, and X.-S. Gao, editors, *Proc. ISSAC '16*, pages 271–278. ACM, 2016.

[19] M. A. Huang and A. J. Rao. Interpolation of sparse multivariate polynomials over large finite fields with applications. In *SODA '96: Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 508–517. Philadelphia, PA, USA, 1996. Society for Industrial and Applied Mathematics.

[20] M. Javadi and M. Monagan. Parallel sparse polynomial interpolation over finite fields. In M. Moreno Maza and J.-L. Roch, editors, *PASCO '10: Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, pages 160–168. ACM Press, 2010.

[21] E. Kaltofen. Computing with polynomials given by straight-line programs I: greatest common divisors. In *STOC '85: Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, pages 131–142. ACM Press, 1985.

[22] E. Kaltofen, Y. N. Lakshman, and J.-M. Wiley. Modular rational sparse multivariate polynomial interpolation. In *ISSAC '90: Proceedings of the International Symposium on Symbolic and Algebraic Computation*, pages 135–139. New York, NY, USA, 1990. ACM Press.

[23] E. Kaltofen and B. M. Trager. Computing with polynomials given by black boxes for their evaluations: greatest common divisors, factorization, separation of numerators and denominators. *J. Symbolic Comput.*, 9(3):301–320, 1990.

[24] E. Kaltofen and L. Yagati. Improved sparse multivariate polynomial interpolation algorithms. In *ISSAC '88: Proceedings of the International Symposium on Symbolic and Algebraic Computation*, pages 467–474. Springer Verlag, 1988.

[25] R. Larrieu. The truncated Fourier transform for mixed radices. In *Proc. ISSAC '17*, pages 261–268. New York, NY, USA, 2017. ACM.

[26] M. Law and M. Monagan. A parallel implementation for polynomial multiplication modulo a prime. In *Proc. of PASCO 2015*, pages 78–86. ACM, 2015.

[27] R. Moenck. Fast computation of GCDs. In *Proc. of the 5th ACM Annual Symposium on Theory of Computing*, pages 142–171. New York, 1973. ACM Press.

[28] H. Murao and T. Fujise. Modular algorithm for sparse multivariate polynomial interpolation and its parallel implementation. *JSC*, 21:377–396, 1996.

[29] R. Prony. Essai expérimental et analytique sur les lois de la dilatabilité des fluides élastiques et sur celles de la force expansive de la vapeur de l'eau et de la vapeur de l'alkool, à différentes températures. *J. de l'École Polytechnique Floréal et Plairial, an III*, 1(cahier 22):24–76, 1795.

[30] D. S. Roche. What can (and can't) we do with sparse polynomials? In C. Arreche, editor, *ISSAC '18: Proceedings of the 2018 ACM International Symposium on Symbolic and Algebraic Computation*, pages 25–30. ACM Press, 2018.

[31] V. Shoup. A new polynomial factorization and its implementation. *J. Symbolic Computation*, 20(4):363–397, 1995.