Implementing the Tangent Graeffe Root Finding Method

Joris van der Hoeven, Michael Monagan

Implementing the tangent Graeffe root finding method***

Joris van der Hoeven[1,2] and Michael Monagan[1]

[1] Dept. of Mathematics, Simon Fraser University, Canada
[2] CNRS, LIX, École polytechnique, France

Abstract. 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 implementation.

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

$$f = \sum_{1 \leqslant i \leqslant t} c_i \boldsymbol{x}^{\boldsymbol{e}_i} \tag{1}$$

of monomials $\boldsymbol{x}^{\boldsymbol{e}_i} = x_1^{e_{1,1}} \cdots x_n^{e_{1,n}}$. 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 [15] and became well known after Ben-Or and Tiwari's seminal paper [2]. It has widely been used in computer algebra, both in theory and in practice; see [16] 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 $\boldsymbol{\alpha}^0, \boldsymbol{\alpha}^1, \ldots, \boldsymbol{\alpha}^{2T-2}$, where $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n) \in \mathbb{K}^n$ and $\boldsymbol{\alpha}^k := (\alpha_1^k, \ldots, \alpha_n^k)$ for all $k \in \mathbb{N}$. The generating function of the evaluations at $\boldsymbol{\alpha}^k$ satisfies the identity

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

where $\Lambda = (1 - \boldsymbol{\alpha}^{\boldsymbol{e}_1} z) \cdots (1 - \boldsymbol{\alpha}^{\boldsymbol{e}_t} z)$ and $N \in \mathbb{K}[z]$ is of degree $< t$. The rational function N/Λ can be recovered from $f(\boldsymbol{\alpha}^0), f(\boldsymbol{\alpha}^1), \ldots, f(\boldsymbol{\alpha}^{2T-2})$ using fast Padé

* *Note:* This paper received funding from NSERC (Canada) and "Agence de l'innovation de défense" (France).

** *Note:* This document has been written using GNU TEXMACS [13].

approximation [4]. For well chosen points $\boldsymbol{\alpha}$, it is often possible to recover the exponents $\boldsymbol{e}_i$ from the values $\boldsymbol{\alpha}^{\boldsymbol{e}_i} \in \mathbb{K}$. If the exponents $\boldsymbol{e}_i$ are known, then the coefficients c_i can also be recovered using fast structured linear algebra [5]. This leaves us with the question how to compute the roots $\boldsymbol{\alpha}^{-\boldsymbol{e}_i}$ 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, and reduce to 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 $s2^l + 1$ 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 [6]. In [10,11], 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 recall the Graeffe transform and the heuristic root finding method based on the tangent Graeffe transform from [10]. 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 twofold. 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). In the longer preprint version of the paper [12], we also show how to generalize our methods to Graeffe transforms of general orders and how to use it in combination with the truncated Fourier transform.

Section 4 is devoted to our new sequential and parallel implementations of the algorithm in C and Cilk C. Our sequential implementation confirms 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 $r, s \geqslant 2$, then we have

$$G_{rs} = G_r \circ G_s = G_s \circ G_r. \tag{3}$$

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*. Now let $P \in \mathbb{K}[z]$ be of the form $P = (z-\alpha_1)\cdots(z-\alpha_d)$ 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 $\alpha_k = r\dfrac{\alpha_k^r}{r\alpha_k^{r-1}}$.

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 $\varnothing$
2. Let $r = 2^N \in 2^{\mathbb{N}}$ be largest such that $r \leqslant (p-1)/(4d)$ and let $s := (p-1)/r$
3. Pick $\tau \in \mathbb{F}_p$ at random and 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. Let ω have order $p-1$ in $\mathbb{F}_p$. Write $\tilde{P} = A + B\epsilon$ and compute $A(\omega^{ir})$, $A'(\omega^{ir})$, and $B(\omega^{ir})$ for $0 \leqslant i < s$
7. If $P(\tau) = 0$, then set $S := \{\tau\}$, else set $S := \varnothing$
8. For $\beta \in \{1, \omega^r, \ldots, \omega^{(s-1)r}\}$ if $A(\beta) = 0$ and $A'(\beta) \neq 0$, set $S := S \cup \{r\beta A'(\beta)/B(\beta) + \tau\}$
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 thus performs $O(\log(p/s))$ 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.

3 Computing Graeffe transforms

3.1 Reminders about discrete Fourier transforms

Assume $n \in \mathbb{N}$ 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

$$\mathrm{DFT}_{\omega^{-1}}((\hat{a}_k)_{0 \leqslant k < n})_i = \sum_{0 \leqslant k < n} \hat{a}_k \omega^{-ik} = \sum_{0 \leqslant j < n} a_j \sum_{0 \leqslant k < n} \omega^{(j-i)k} = n a_i. \quad (4)$$

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 it is well known [7] that

$$\hat{a}_{k_2 n_1 + k_1} = \mathrm{DFT}_{\omega^{n_1}} \left(\left(\omega^{i_2 k_1} \mathrm{DFT}_{\omega^{n_2}}((a_{i_1 n_2 + i_2})_{0 \leqslant i_1 < n_1})_{k_1} \right)_{0 \leqslant i_2 < n_2} \right)_{k_2}. \quad (5)$$

This means 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 $\mathsf{F}_{\mathbb{K}}(rn) \sim r \mathsf{F}_{\mathbb{K}}(n)$.

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 $\deg(AB) < n$, we may then compute the product AB using

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

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 3. 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 (5), 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

$$G(P)(z^2) = \mathrm{DFT}_\omega^{-1}(\mathrm{DFT}_\omega(P(z))\,\mathrm{DFT}_\omega(P(-z))). \tag{6}$$

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

$$\mathrm{DFT}_\omega(P(-z))_k = P(-\omega^k) = P(\omega^{(k+n) \bmod 2n}) = \mathrm{DFT}_\omega(P(z))_{(k+n) \bmod 2n}\,, \tag{7}$$

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 $k = 0, \ldots, n-1$. Plugging this into (6), we conclude that

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

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 $(\hat{P}_k)_{0 \leqslant k < 2n} := \mathrm{DFT}_\omega(P)$
2. For $k = 0, \ldots, n-1$, compute $\hat{G}_k := \hat{P}_k \hat{P}_{k+n}$
3. Return $\mathrm{DFT}_{\omega^2}^{-1}((\hat{G}_k)_{0 \leqslant k < n})$

Proposition 1. *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 $\mathsf{F}_{\mathbb{K}}(2n) = 2\mathsf{F}_{\mathbb{K}}(n) + O(n)$. Step 2 can be done in $O(n)$. Step 3 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 4. 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 [10]. Note that the best known algorithm for squaring 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 (3), Graeffe transforms of power of two orders 2^m can be computed using

$$G_{2^m}(P) = \left(G \circ \overset{m\times}{\cdots} \circ G\right)(P). \tag{8}$$

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

$$\mathrm{DFT}_{\omega}(G(P))_{2k} = \mathrm{DFT}_{\omega^2}(G(P))_k = \hat{G}_{2k} \tag{9}$$

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 (5) for $n_1 = 2$ and $n_2 = n$, we have

$$\mathrm{DFT}_{\omega}(G(P))_{2k+1} = \mathrm{DFT}_{\omega^2}((\omega^i G(P)_i)_{0\leqslant i<n})_k \tag{10}$$

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 $(\hat{Q}_k)_{0\leqslant k<n} = \mathrm{DFT}_{\omega^2}(P)$
Output: $G(P)$ and $\mathrm{DFT}_{\omega^2}(G(P))$

---

1. Set $(\hat{P}_{2k})_{0\leqslant k<n} := (\hat{Q}_k)_{0\leqslant k<n}$
2. Set $(\hat{P}_{2k+1})_{0\leqslant k<n} := \mathrm{DFT}_{\omega^2}((\omega^i P_i)_{0\leqslant i<n})$
3. For $k = 0, \ldots, n-1$, compute $\hat{G}_k := \hat{P}_k \hat{P}_{k+n}$
4. Return $\mathrm{DFT}^{-1}_{\omega^2}((\hat{G}_k)_{0\leqslant k<n})$ and $(\hat{G}_k)_{0\leqslant k<n}$

Proposition 2. *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 5. In [10], Graeffe transforms of order 2^m were directly computed using the formula (8), using $\sim 4m\mathsf{F}_{\mathbb{K}}(n)$ operations in $\mathbb{K}$, which is twice as slow as the new algorithm.

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 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 [17]. For our input $P(z)$, with d distinct linear factors, Shoup uses the Cantor–Zassenhaus equal degree factorization method. The average complexity of TG is $O(\mathsf{M}(d)(\log(p/s)+\log d))$ and of CZ is $O(\mathsf{M}(d) \log p \log d)$.

	Our sequential 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 for $p = 3 \cdot 29 \cdot 2^{56} + 1$ and using $s \in [2d, 4d)$.

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), which obtains about 69% of the roots. The time in column "total" is the total time for the TG 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 timings are for Magma's `Factorization` command. The timings for Magma version V2.25-3 suggest that Magma's CZ implementation involves a subalgorithm with quadratic asymptotic complexity. Indeed it turns out that the author of the code implemented all of the sub-quadratic polynomial arithmetic

correctly, as demonstrated 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).

We also wanted to attempt a parallel implementation. To do this we used the MIT Cilk C compiler from [8]. Cilk provides a simple fork-join model of parallelism. Unlike the CZ algorithm, TG has no gcd computations that are hard to parallelize. We present some initial parallel timing data in Table 2. The timings in parentheses are parallel timings for 8 cores.

	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]. For step 5 we use Algorithm 3. It has complexity $O(\mathsf{M}(d)\log\frac{p}{s})$. 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 $O(\mathsf{M}(t)\log t)$ product tree multiplication algorithm [9]. 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 90% 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 [14]. 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 $d = 2^{25} - 1$.

References

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. 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.
5. 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.
6. D. G. Cantor and H. Zassenhaus. A new algorithm for factoring polynomials over finite fields. *Math. Comp.*, 36(154):587–592, 1981.
7. J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Computat.*, 19:297–301, 1965.
8. 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.
9. J. von zur Gathen and J. Gerhard. *Modern computer algebra*. Cambridge University Press, New York, 3rd edition, 2013.
10. 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.
11. B. Grenet, J. van der Hoeven, and G. Lecerf. Deterministic root finding over finite fields using Graeffe transforms. *AAECC*, 27(3):237–257, 2016.
12. J. van der Hoeven and M. Monagan. Implementing the tangent Graeffe root finding method. Technical Report, HAL, 2020. `http://hal.archives-ouvertes.fr/hal-02525408`.
13. J. van der Hoeven et al. GNU TeXmacs. `http://www.texmacs.org`, 1998.
14. M. Law and M. Monagan. A parallel implementation for polynomial multiplication modulo a prime. In *Proc. of PASCO 2015*, pages 78–86. ACM, 2015.
15. 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.
16. 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.
17. V. Shoup. A new polynomial factorization and its implementation. *J. Symbolic Computation*, 20(4):363–397, 1995.