Implementation of a Near-Optimal Complex Root Clustering Algorithm

Rémi Imbach[1] *, Victor Y. Pan[2] **, and Chee Yap[3] ***

[1] TU Kaiserslautern
Email: imbach@mathematik.uni-kl.de
www.mathematik.uni-kl.de/en/agag/members/
[2] City University of New York
Email: victor.pan@lehman.cuny.edu
http://comet.lehman.cuny.edu/vpan/
[3] Courant Institute of Mathematical Sciences
New York University, USA
Email: yap@cs.nyu.edu
www.cs.nyu.edu/yap/

Abstract. We describe `Ccluster`, a software for computing natural ε-clusters of complex roots in a given box of the complex plane. This algorithm from Becker et al. (2017) is near-optimal when applied to the benchmark problem of isolating all complex roots of an integer polynomial. It is one[4] of the first implementations of a near-optimal algorithm for complex roots. We describe some low level techniques for speeding up the algorithm. Its performance is compared with the well-known `MPSolve` library and `Maple`.

1 Introduction

The problem of root finding for a polynomial $f(z)$ is a classical problem from antiquity, but remains the subject of active research to the present [6]. We consider a classic version of root finding:

Local root isolation problem:
Given: *a polynomial* $f(z) \in \mathbb{C}[z]$, *a box* $B_0 \subseteq \mathbb{C}$, $\varepsilon > 0$.
Output: *a set* $\{\Delta_1, \ldots, \Delta_k\}$ *of pairwise-disjoint discs of radius* $\leq \varepsilon$, *each containing a unique root of* $f(x)$ *in* B_0.

It is local because we only look for roots in a locality, as specified by B_0. The local problem is useful in applications (especially in geometric computation) where

* Rémi's work has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No. 676541.
** Victor's work is supported by NSF Grants # CCF-1116736 and # CCF-1563942 and by PSC CUNY Award 698130048.
*** Chee's work is supported by NSF Grants # CCF-1423228 and # CCF-1564132.

[4] Irina Voiculescu informed us that her student Dan-Andrei Gheorghe has independently implemented the same algorithm in a Masters Thesis Project (May 18, 2017) at Oxford University.

we know where to look for the roots of interest. There are several variants of this problem: in the **global version**, we are not given B_0, signifying that we wish to find all the roots of f. The global version is easily reduced to the local one by specifying a B_0 that contains all roots of f. If we omit ε, it amounts to setting $\varepsilon = \infty$, representing the pure isolation problem.

Our main interest is a generalization of root isolation, to the lesser-studied problem of root clustering [10, 12, 8]. It is convenient to introduce two definitions: for any set $S \subseteq \mathbb{C}$, let $Z_f(S)$ denote the set of roots of f in S, and let $\#_f(S)$ count the total multiplicity of the roots in $Z_f(S)$. Typically, S is a disc or a box. For boxes and discs, we may write kS (for any $k > 0$) to denote the dilation of S by factor k, keeping the same center. The following problem was introduced in [17]:

> **Local root clustering problem**:
> **Given:** *a polynomial $f(z)$, a box $B_0 \subseteq \mathbb{C}$, $\varepsilon > 0$.*
> **Output:** *a set of pairs $\{(\Delta_1, m_1), \ldots, (\Delta_k, m_k)\}$ where*
>
> - *Δ_i's are pairwise-disjoint discs of radius $\leq \varepsilon$,*
> - *$m_i = \#_f(\Delta_i) = \#_f(3\Delta_i)$ for all i, and*
> - *$Z_f(B_0) \subseteq \bigcup_{i=1}^{k} Z_f(\Delta_i)$.*

This generalization of root isolation is necessary when we consider polynomials whose coefficients are non-algebraic (or when $f(z)$ is an analytic function, as in [17]). The requirement that $\#_f(\Delta_i) = \#_f(3\Delta_i)$ ensures that our output clusters are **natural** [1]; a polynomial of degree d has at most $2d-1$ natural clusters (see [17, Lemma 1]). The local root clustering algorithm for analytic functions of [17] has termination proof, but no complexity analysis. By restricting $f(z)$ to a polymomial, Becker et al. [2] succeeded in giving an algorithm and also its complexity analysis based on the geometry of the roots. When applied to the **benchmark problem**, where $f(z)$ is an integer polynomial of degree d with L-bit coefficients, the algorithm can isolate all the roots of $f(z)$ with bit complexity $\widetilde{O}(d^2(L+d))$. Pan [13] calls such bounds **near-optimal** (at least when $L \geq d$). The clustering algorithm studied in this paper comes from [1], which in turn is based on [2]. Previously, the Pan-Schönhage algorithm has achieved near-optimal bounds with divide-and-conquer methods [13], but [2, 1] was the first *subdivision* algorithm to achieve the near-optimal bound for complex roots. For real roots, Sagraloff-Mehlhorn [16] had earlier achieved near-optimal bound via subdivision.

Why the emphasis on "subdivision"? It is because such algorithms are implementable and quite practical (e.g., [15]). Thus the near-optimal real subdivision algorithm of [16] was implemented shortly after its discovery, and reported in [11] with excellent results. In contrast, all the asymptotically efficient root algorithms (not necessarily near-optimal) based on divide-and-conquer methods of the last 30 years have never been implemented; a proof-of-concept implementation of Schönhage's algorithm was reported in Gourdon's thesis [9]). Computer algebra systems mainly rely on algorithms with a priori guarantees of correctness. But in practice, algorithms without such guarantees are widely used. For complex root isolation, one of the most highly regarded multiprecision software is `MPSolve` [3]. The original algorithm in

MPSolve was based on Erhlich-Aberth (EA) iteration; but since 2014, a "hybrid" algorithm [4] was introduced. It is based on the secular equation, and combines ideas from EA and eigensolve [7]. These algorithms are inherently global solvers (they must approximate *all* roots of a polynomial simultaneously). Another theoretical limitation is that the global convergence of these methods is not proven.

In this paper, we give a preliminary report about Ccluster, our implementation of the root clustering algorithm from [1].

To illustrate the performance for the local versus global problem, consider the Bernoulli polynomials $\mathtt{Bern}_d(z) := \sum_{k=0}^{d} \binom{d}{k} b_{d-k} z^k$ where b_i's are the Bernoulli numbers. Figure 1(Left) shows the graphical output of Ccluster for $\mathtt{Bern}_{100}(z)$. Table 1 has four timings τ_X (for $X = \ell, g, u, s$) in seconds: τ_ℓ is the time for solving the local problem over a box $B_0 = [-1,1]^2$; τ_g is the time for the global problem over the box $B_0 = [-150,150]^2$ (which contains all the roots). The other two timings from MPSolve (τ_u for unisolve, τ_s for secsolve) will be explained later. For each instance, we also indicate the numbers of solutions (#Sols) and clusters (#Clus). When #Sols equals #Clus, we know the roots are isolated. Subdivision algorithms like ours naturally solve the local problem, but MPSolve can only solve the global problem. Table 1 shows that MPSolve remains unchallenged for the global problem. But in applications where locality can be exploited, local methods may win, as seen in the last two rows of the table. The corresponding time for Maple's fsolve is also given; fsolve is not a guaranteed algorithm and may fail.

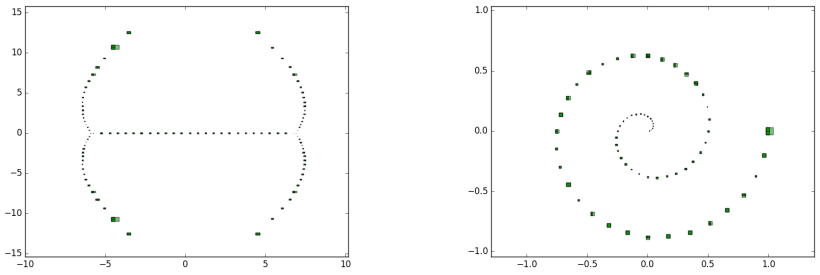

Fig. 1. Left: the connected components isolating all roots of the Bernoulli polynomial of degree 100. **Right:** the connected components isolating all roots of the Spiral polynomial of degree 64.

1.1 Overview of Paper

In Section 2, we describe the experimental setup for Ccluster. Sections 3-5 describe some techniques for speeding up the basic algorithm. We conclude with Section 6.

	Ccluster local ($B_0=[-1,1]^2$)			Ccluster global ($B_0=[-150,150]^2$)			unisolve	secsolve	fsolve
d	(#Sols:#Clus)	(depth:size)	τ_ℓ (s)	(#Sols:#Clus)	(depth:size)	τ_g (s)	τ_u (s)	τ_s (s)	τ_f (s)
64	(4:4)	(9:164)	0.12	(64:64)	(17:1948)	2.10	0.13	0.01	0.1
128	(4:4)	(9:164)	0.34	(128:128)	(16:3868)	9.90	0.55	0.05	6.84
191	(5:5)	(9:196)	0.69	(191:191)	(17:5436)	32.5	2.29	0.16	50.0
256	(4:4)	(9:164)	0.96	(256:256)	(17:7300)	60.6	3.80	0.37	>1000
383	(5:5)	(9:196)	2.06	(383:383)	(17:11188)	181	>1000	1.17	>1000
512	(4:4)	(9:164)	2.87	(512:512)	(16:14972)	456	>1000	3.63	>1000
767	(5:5)	(9:196)	6.09	(767:767)	(17:22332)	1413	>1000	10.38	>1000

Table 1. Bernoulli Polynomials with five timings: local (τ_ℓ), global (τ_ℓ), unisolve (τ_ℓ), secsolve (τ_ℓ) and Maple's fsolve(τ_f).

2 Implementation and Experiments

The main implementation of Ccluster is in C language. We have an interface for Julia[5]. We based our big number computation on the arb[6] library. The arb library implements ball arithmetic for real numbers, complex numbers and polynomials with complex coefficients. Each arithmetic operation is carried out with error bounds.

Test Suite We consider 7 families of polynomials, classic ones as well as some new ones constructed to have interesting clustering or multiple root structure.

(F1) The Bernoulli polynomial $\texttt{Bern}_d(z)$ of degree d was described earlier in the introduction.

(F2) The Mignotte polynomial $\texttt{Mign}_d(z;a) := z^d - 2(2^a z-1)^2$ for a positive integer a, has two roots whose separation is near the theoretical minimum separation bound.

(F3) The Wilkinson polynomials $\texttt{Wilk}_d(z) := \prod_{k=1}^{d}(z-k)$.

(F4) The Spiral Polynomial $\texttt{Spir}_d(z) := \prod_{k=1}^{d}\left(z - \frac{k}{d}e^{4k\mathbf{i}\pi/n}\right)$. See Figure 1(Right) for $\texttt{Spir}_{64}(z)$.

(F5) Wilkinson Multiple: $\texttt{WilkMul}_{(D)}(z) := \prod_{k=1}^{D}(z-k)^k$. $\texttt{WilkMul}_{(D)}(z)$ has degree $d = D(D+1)/2$ where the root $z=k$ has multiplicity k (for $k=1,\ldots,D$).

(F6) Mignotte Cluster: $\texttt{MignClu}_d(z;a,k) := x^d - 2(2^a z-1)^k(2^a z+1)^k$. This polynomial has degree d (assuming $d \geq 2k$) and has a cluster of k roots near 2^{-a} and a cluster of k roots near -2^{-a}.

(F7) Nested Cluster: $\texttt{NestClu}_{(D)}(z)$ has degree $d = 3^D$ and is defined by induction on D: $\texttt{NestClu}_{(1)}(z) := z^3 - 1$ with roots $\omega, \omega^2, \omega^3 = 1$ where $\omega = e^{2\pi\mathbf{i}/3}$. Inductively, if the roots of $\texttt{NestClu}_{(D)}(z)$ are $\{r_j : j = 1,\ldots,3^D\}$, then we define $\texttt{NestClu}_{(D+1)}(z) := \prod_{j=1}^{3^D}\left(z - r_j - \frac{\omega}{16^D}\right)\left(z - r_j - \frac{\omega^2}{16^D}\right)\left(z - r_j - \frac{1}{16^D}\right)$ See Figure 2 for the natural ε-clusters of $\texttt{NestClu}_{(3)}(z)$.

[5] https://julialang.org/. Download our code in https://github.com/rimbach/Ccluster.
[6] http://arblib.org/. Download our code in https://github.com/rimbach/Ccluster.jl.

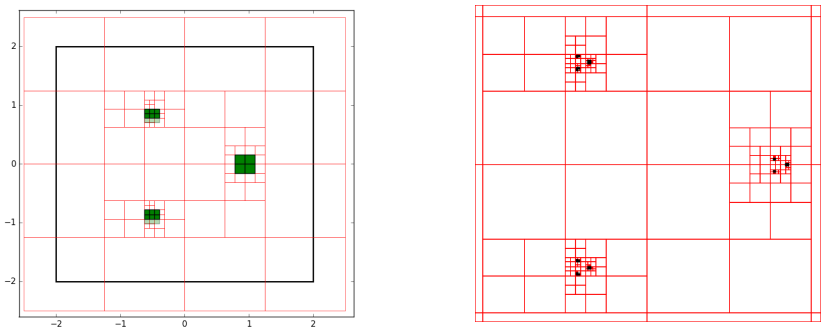

Fig. 2. Left: 3 clusters of $\texttt{NestClu}_{(3)}$ found with $\varepsilon = 1$. **Right:** Zoomed view of 9 clusters of $\texttt{NestClu}_{(3)}$ found with $\varepsilon = \frac{1}{10}$. **Note:** The initial box is in thick lines; the thin lines show the subdivisions tree.

Timing Running times are sequential times on a Intel(R) Core(TM) i3 CPU 530 @ 2.93GHz machine with linux. `Ccluster` implements the algorithm described in [1] with differences coming from the improvements described in Sections 3-5 below. Unless explicitly specified, the value of ε for `Ccluster` is set to 2^{-53}; roughly speaking, it falls back to asking for 15 guaranteed decimal digits.

`MPSolve` For external comparison, we use `MPSolve`. It was shown (circa 2000) to be superior to major software such as `Maple` or `Mathematica` [3]. There are two root solvers in `MPSolve`: the original `unisolve` [3] which is based on the Ehrlich-Aberth iteration and the new hybrid algorithm called `secsolve` [4]. These are called with the commands `mpsolve -au -Gi -o`γ `-j1` and `mpsolve -as -Gi -o`γ `-j1` (respectively). `-Gi` means that `MPSolve` tries to find for each root a unique complex disc containing it, such that Newton iteration is guaranteed to converge quadratically toward the root starting from the center of the disc. `-o`γ means that $10^{-\gamma}$ is used as an escape bound, *i.e.,* the algorithm stops when the complex disc containing the root has radius less that $10^{-\gamma}$, regardless of whether it is isolating or not. Unless explicitly specified, we set $\gamma = 16$. `-j1` means that the process is not parallelized. Although `MPSolve` does not do general local search, it has an option to search only within the unit disc. This option does not seem to lead to much improvement.

3 Improved Soft Pellet Test

The key predicate in [1] is a form of Pellet test denoted $\widetilde{T}_k^G(\Delta, k)$ (with implicit $f(z)$). This is modified in Figure 3 by adding an outer while-loop to control the number of Graeffe-Dandelin iterations. We try to get a definite decision (i.e., anything other than a **unresolved**) from the soft comparison for the current Graeffe iteration. This is done by increasing the precision L for approximating the coefficients of $\tilde{f}$ in the innermost while-loop. Thus we have two versions of our algorithm: (V1) uses the original $\widetilde{T}_k^G(\Delta, k)$ in [1], and (V2) uses the modified form in Figure 3. Let τV1 and

$\widetilde{T}_k^G(\Delta,k)$ ◁ *$f(z)$ is implicit argument*
Output: $res \in \{-1,0\ldots,k\}$
ASSERT: if $res \geq 0$, then $\#_f(\Delta) = res$
$L \leftarrow 53$, $d \leftarrow \deg(f)$, $N \leftarrow 4+\lceil \log_2(1+\log_2(d))\rceil$, $i \leftarrow 0$
$\tilde{f} \leftarrow$getApproximation(f, L)
$\tilde{f} \leftarrow$TaylorShift($\tilde{f}$, Δ)
While $i \leq N$
Let $\tilde{f}$ be the i-th Graeffe iteration of $\tilde{f}$
$res \leftarrow 0$
While $res \leq k$
$j \leftarrow$ `IntCompare`($abs(res$-th coeff of $\tilde{f})$, sum of abs values of other coeffs, 2^{-L})
While $j =$ **unresolved**
$L \leftarrow 2L$
$\tilde{f} \leftarrow$getApproximation(f, L)
$\tilde{f} \leftarrow$TaylorShift($\tilde{f},\Delta$)
Let $\tilde{f}$ be i-th Graeffe iteration of $\tilde{f}$
$j \leftarrow$`IntCompare`($abs(res$-th coeff of $\tilde{f})$, sum of abs values of other coeffs, 2^{-L})
If $j =$ **true** then **Return** res
$res \leftarrow res+1$
$i \leftarrow i+1$
Return -1

Fig. 3. $T_k test(P,\Delta,k)$

τV2 be timings for the 2 versions. Table 2 shows the time τV1 (in seconds) and the ratio τV1/τV2. We see that (V2) achieves a consistent 2.3 to 3-fold speed up.

	V1		V2		V3	
	(n1, n2, n3)	τV1	(n1, n2, n3)	τV1/τV2	(n1, n2, n3)	τV1/τV3
`Bern`$_{64}(z)$	(2308,686,20223)	19.6	(2308,686,6028)	2.84	(2308,8,2291)	7.06
`Mign`$_{64}(z;14)$	(2060,622,18018)	17.3	(2060,622,5326)	3.03	(2060,20,2080)	7.68
`Wilk`$_{64}(z)$	(2148,674,18053)	23.6	(2148,674,5692)	2.74	(2148,0,2140)	7.23
`Spir`$_{64}(z)$	(2512,728,22176)	22.2	(2512,728,6596)	2.39	(2512,15,2670)	4.46
`WilkMul`$_{(11)}(z)$	(724,202,6174)	9.69	(724,202,2684)	2.30	(724,18,2065)	3.37
`MignClu`$_{64}(z;14,3)$	(2092,618,18515)	20.0	(2092,618,5600)	3.00	(2092,12,2481)	6.57
`NestClu`$_{(4)}(z)$	(3532,1001,30961)	90.2	(3532,1001,9654)	3.09	(3532,24,4588)	6.81

Table 2. Solving within the initial box $[-50,50]^2$ with $\varepsilon = 2^{-53}$ with versions (V1), (V2) and (V3) of `Ccluster`. n1: number of discarding tests. n2: number of discarding tests returning -1 (inconclusive). n3: total number of Graeffe iterations. τV1 (resp. τV2, τV3): sequential time for V1 (resp. V2, V3) in seconds.

In (V2), as in [1], we use $\widetilde{T}_0^G(\Delta)$ to prove that a box B has no root. We propose a new version (V3) that uses $\widetilde{T}_*^G(\Delta)$ instead of $\widetilde{T}_0^G(\Delta)$ to achieve this goal: instead of just showing that B has no root, it upper bounds $\#_f(B)$. Although counter-intuitive, this yields a substantial improvement because it led to fewer Graeffe iterations overall. The timing for (V3) is τV3, but we display only the ratio τV1/τV3 in the last

column of Table 2. This ratio shows that (V3) enjoys a 3.3-7.7 fold speedup. Comparing $n3$ for (V2) and (V3) explains this speedup.

4 Filtering

A technique for speeding up the evaluation of predicates is the idea of filters (e.g., [5]). The various Pellet tests can be viewed as a box predicate C that maps a box $B \subseteq \mathbb{C}$ to a value[7] in $\{\textbf{true}, \textbf{false}\}$. If C^- is another box predicate with property that $C^-(B) = \textbf{false}$ implies $C(B) = \textbf{false}$, we call C^- a **falsehood filter**. If C^- is efficient relatively to C, and "efficacious" (informally, $C(B) = \textbf{false}$ is likely to yield $C^-(B) = \textbf{false}$), then it is useful to first compute $C^-(B)$. If $C^-(B) = \textbf{false}$, we do not need to compute $C(B)$. `Ccluster` uses the predicate C_0 to prove that box B contains no root of f. It is defined as follows: $C_0(B)$ is **true** if $\widetilde{T}^G_*(\Delta_B)$ returns 0 (then B contains no root of f) and is **false** if $\widetilde{T}^G_*(f, \Delta_B)$ returns -1 or $k > 0$ (then B may contain some roots of f). We next present a falsehood filter $C_0^-(B)$ for C_0.

Let f_Δ denote the Taylor shift of f in Δ, $f_\Delta^{[i]}$ its i-th Graeffe iterate, $(f_\Delta^{[i]})_j$ the j-th coefficient of $f_\Delta^{[i]}$, and $|f_\Delta^{[i]}|_j$ the absolute value of the j-th coefficient. Let d be the degree of f. The assertions below are direct consequences of the classical test of Pellet (see eq. 2 in App. 6) and justify the correctness of our filter:

$(A1)$ if $|f_\Delta^{[N]}|_0 \leq |f_\Delta^{[N]}|_d$ then $\widetilde{T}^G_*(f, \Delta)$ returns -1 or $k > 0$,

$(A2)$ if $|f_\Delta^{[N]}|_0 \leq |f_\Delta^{[N]}|_1 + |f_\Delta^{[N]}|_d$ then $\widetilde{T}^G_*(f, \Delta)$ returns -1 or $k > 0$.

Our C_0^- filter computes $|f_\Delta^{[N]}|_0$, $|f_\Delta^{[N]}|_1$ and $|f_\Delta^{[N]}|_d$ and makes the comparisons $(A1)$ or $(A2)$ using `SoftCompare`. $|f_\Delta^{[N]}|_0$ and $|f_\Delta^{[N]}|_d$ are respectively computed as $(|f_\Delta|_0)^{2^N}$ and $(|f_\Delta|_d)^{2^N}$. $|f_\Delta^{[N]}|_1$ can be computed with the following well known formula:

$$(f_\Delta^{[i+1]})_k = (-1)^k((f_\Delta^{[i]})_k)^2 + 2\sum_{j=0}^{k-1}(-1)^j(f_\Delta^{[i]})_j(f_\Delta^{[i]})_{2k-j} \qquad (1)$$

Obtaining $|f_\Delta^{[N]}|_1$ with eq. (1) requires to know $2^{N-1}+1$ coefficients of $f_\Delta^{[1]}$, $2^{N-2}+1$ coefficients of $f_\Delta^{[2]}, \ldots$, and finally $3 = 2^1 + 1$ coefficients of $f_\Delta^{[N-1]}$. In particular, it requires to compute entirely the iterations $f_\Delta^{[i]}$ such that $2^{N-i} \leq d$, and it is possible to do it more efficiently that with eq. (1) (for instance with the formula given in definition 2 of [2]). In summary, the C_0^- filter computes first f_Δ, then $|f_\Delta^{[N]}|_0$ and $|f_\Delta^{[N]}|_d$ and returns **false** if the hypothesis of $(A1)$ is verified. Then it computes $f_\Delta^{[i]}$ for i satisfying $2^{N-i} \leq \frac{d}{4}$ before the approach described above and $|f_\Delta^{[N]}|_1$ is obtained. Finally, **false** is returned if the hypothesis of $(A2)$ is verified, and **true** otherwise.

In practice, C_0^- is performed within $\widetilde{T}^G_*(f, \Delta_B)$. Incorporating this into Version (V3), we obtain (V4) and the speed up can be seen in Table 3. We note filtering with C_0^- becomes more effective as degree grows and this is because one has $2^{N-i} \leq \frac{d}{4}$ for smaller i (recall that $N = 4 + \lceil \log_2(1 + \log_2(d)) \rceil$).

[7] We treat two-valued predicates for simplicity; the discussion could be extended to predicates (like $\widetilde{T}^G_*$) which returns a finite set of values.

		V3		V4	
		n3	τV3	n3	τV3/τV4
$\texttt{Bern}_d(z)$	$d=64$	2291	2.61	2084	1.08
	$d=128$	4496	14.5	3983	1.13
	$d=256$	8847	94.5	7714	1.19
	$d=512$	15983	620	11664	1.42
$\texttt{Mign}_d(z;a)$	$(d,a)=(64,14)$	2080	2.41	1808	1.22
	$(d,a)=(128,14)$	3899	12.1	3257	1.21
	$(d,a)=(256,14)$	7605	88.3	6339	1.33
	$(d,a)=(512,14)$	15227	674	10405	1.57
$\texttt{Wilk}_d(z)$	$d=64$	2140	3.27	1958	1.05
	$d=128$	2240	10.0	1942	1.09
	$d=256$	2414	36.6	2108	1.21
	$d=512$	2557	129	1841	1.43
$\texttt{Spir}_d(z)$	$d=64$	2670	4.43	2364	1.08
	$d=128$	5090	28.8	4405	1.07
	$d=256$	9746	182	8529	1.10
	$d=512$	19159	1340	14786	1.19
$\texttt{WilkMul}_{(D)}(z)$	$(D,d)=(11,66)$	2065	2.87	1818	1.14
	$(D,d)=(12,78)$	2313	3.95	2053	1.12
	$(D,d)=(13,91)$	2649	5.89	2336	1.18
	$(D,d)=(14,105)$	2892	8.56	2537	1.29
$\texttt{MignClu}_d(z;a,k)$	$(d,a,k)=(64,14,3)$	2481	2.94	2145	1.13
	$(d,a,k)=(128,14,3)$	4166	14.4	3555	1.16
	$(d,a,k)=(256,14,3)$	7658	86.0	6523	1.27
	$(d,a,k)=(512,14,3)$	15044	650	10472	1.63
$\texttt{NestClu}_{(D)}(z)$	$(D,d)=(4,27)$	1628	0.77	1459	1.07
	$(D,d)=(5,81)$	4588	13.2	4085	1.12
	$(D,d)=(6,243)$	13056	358	11824	1.26

Table 3. Solving within the initial box $[-50,50]^2$ with $\varepsilon = 2^{-53}$ with versions (V3), (V4) of `Ccluster`. n3: number of Graeffe iterations. τV3 and τV4: sequential time in seconds.

5 Escape Bound

The ε parameter is usually understood as the precision desired for roots. But we can also view it as an escape bound for multiple roots as follows: we do not refine a disc that contains a simple root, even if its radius is $\geq \varepsilon$. But for clusters of size greater than one, we only stop when the radius is $< \varepsilon$. `MPSolve` has a similar option. This variant of (V4) is denoted (V4'). We see from Table 4 that (V4') gives a modest improvement (up to 25% speedup) over (V4) when $-\lg\varepsilon = 53$. This improvement generally grows with $-\log\varepsilon$ (but $\texttt{WilkMul}_{(11)}(z)$ shows no difference).

	(V4)			(V4')		
ε:	2^{-53}	2^{-530}	2^{-5300}	2^{-53}	2^{-530}	2^{-5300}
	τ53 (s)	τ530/τ53	τ5300/τ53	τ53 (s)	τ530/τ53	τ5300/τ53
$\texttt{Bern}_{64}(z)$	2.42	1.26	4.22	1.99	0.94	0.94
$\texttt{Mign}_{64}(z;14)$	1.97	1.63	4.56	1.61	1.45	1.38
$\texttt{Wilk}_{64}(z)$	3.22	1.10	2.16	2.91	0.96	1.01
$\texttt{Spir}_{64}(z)$	4.09	1.33	5.25	3.05	0.95	0.95
$\texttt{WilkMul}_{(11)}(z)$	2.51	1.12	2.03	2.50	1.13	1.98
$\texttt{MignClu}_{64}(z;14,3)$	2.60	1.89	4.15	2.20	1.70	1.80
$\texttt{NestClu}_4(z)$	11.9	1.08	2.67	10.4	1.00	0.99

Table 4. Solving within the box $[-50,50]^2$ with versions (V4) and (V4') of `Ccluster` with three values of ε. τ53 (resp. τ530, τ5300): sequential time for (V4) and (V4') in seconds.

6 Conclusion

Implementing subdivision algorithms is relatively easy but achieving state-of-art performance requires much optimization and low-level development. This paper explores several such techniques. We do well compared to `fsolve` in `Maple`, but the performance of `MPSolve` is superior to the global version of `Ccluster`. But `Ccluster` can still shine when looking for local roots or when ε is large.

References

1. R. Becker, M. Sagraloff, V. Sharma, J. Xu, and C. Yap. Complexity analysis of root clustering for a complex polynomial. In *41st ISSAC*, pp. 71–78, 2016.
2. R. Becker, M. Sagraloff, V. Sharma, and C. Yap. A near-optimal subdivision algorithm for complex root isolation based on Pellet test and Newton iteration. *JSC*, 86:51–96, May 2018.
3. D. A. Bini and G. Fiorentino. Design, analysis, and implementation of a multiprecision, polynomial rootfinder. *Numerical Algorithms*, 23:127–173, 2000.
4. D. A. Bini and L. Robol. Solving secular and polynomial equations: A multiprecision algorithm. *J. Computational and Applied Mathematics*, 272:276–292, 2014.
5. H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Applied Math.*, 109(1-2):25–47, 2001.
6. I. Z. Emiris, V. Y. Pan, and E. P. Tsigaridas. Algebraic algorithms. In T. Gonzalez, J. Diaz-Herrera, and A. Tucker, editors, *Computing Handbook, 3rd Edition: Computer Science and Software Engineering*, pages 10: 1–30. Chapman and Hall/CRC, 2014.
7. S. Fortune. An iterated eigenvalue algorithm for approximating roots of univariate polynomials. *J. Symbolic Computation*, 33(5):627–646, 2002.
8. M. Giusti, G. Lecerf, B. Salvy, and J.-C. Yakoubsohn. On location and approximation of clusters of zeros of analytic functions. *Found.Comp.Math.*, 5(3):257–311, July 2005.
9. X. Gourdon. *Combinatoire, Algorithmique et Géométrie des Polynômes*. PhD thesis, École polytechnique, 1996.
10. V. Hribernig and H. J. Stetter. Detection and validation of clusters of polynomial zeros. *J. Symbolic Computation*, 24(6):667–682, 1997.
11. A. Kobel, F. Rouillier, and M. Sagraloff. Computing real roots of real polynomials ... and now for real! In *41st ISSAC*, pages 303–310, 2016. July 19-22, Waterloo, Canada.
12. X.-M. Niu, T. Sakurai, and H. Sugiura. A verified method for bounding clusters of zeros of analytic functions. *J. Comput. Appl. Math.*, 199(2):263–270, Feb. 2007.
13. V. Y. Pan. Univariate polynomials: Nearly optimal algorithms for numerical factorization and root-finding. *J. Symb. Comput.*, 33(5):701–733, 2002.
14. Q. I. Rahman and G. Schmeisser. *Analytic Theory of Polynomials*. Oxford Press, 2002.
15. F. Rouillier and P. Zimmermann. Efficient isolation of [a] polynomial's real roots. *J. Computational and Applied Mathematics*, 162:33–50, 2004.
16. M. Sagraloff and K. Mehlhorn. Computing real roots of real polynomials. *J. Symbolic Computation*, 73:46–86, 2016.
17. C. Yap, M. Sagraloff, and V. Sharma. Analytic root clustering: A complete algorithm using soft zero tests. In P. Bonizzoni, V. Brattka, and B. Lowe, editors, *Computability in Europe (CiE2013)*, vol. 7921 of *LNCS*, pages 434–444, Heidelberg, 2013. Springer.

APPENDIX A. Algorithm Overview

This Appendix is temporarily provided for the Referee's convenient only

We briefly review the `Ccluster` algorithm and its underlying theory. The main computational paradigm is that of subdivision: given a box $B \subseteq \mathbb{C}$, we subdivide B into four congruent subboxes. We assume square boxes only. If this subdivision process is iterated, all boxes B will eventually be small enough to make decisions about presense/absence of roots in B. In particular, we have a box predicate, $P_0(B)$ with the following property: *if $P_0(B)$ holds, then B has no roots of $f(z)$; otherwise $2B$ contains some root.* This "P_0" is a **Pellet test** as explained below. In our subdivision algorithms, we used the exclusion and inclusion predicates, $C_0(B)$ and $C_1(B)$. If $C_0(B)$ (resp., $C_1(B)$) holds, then B has no roots (resp., has some root). But the failure of these predicates yields no information. But $P_0(B)$ yields information whether it is true or not. It is an exclusion test (for B) if it succeeds, and an inclusion test (for $2B$) when it fails.

If subdivision were the only method of decomposition, we could organize our search in the standard subdivision tree, rooted at B_0. But the regularity of subdivision is broken in our algorithm because of Newton iteration: if a Newton step succeeds, box B is replaced by a subbox B' that could be arbitrarily smaller than B. To organize such irregularities, we introduce a higher level concept: **components**: this is a connected set $C \subseteq \mathbb{C}$ obtained as the union of a set $\mathscr{S} = \mathscr{S}_C$ of boxes; the boxes in $\mathscr{S}$ are called the **constituent boxes** of C, and they all have the same width. We can subdivide C by subdividing each constituent box B of C as before: apply P_0 to the subdivided boxes, discarding any B' if $P_0(B')$ succeeds, and reorganizing the remaining boxes into $k \geq 0$ new connected components, denoted: $C \to (C_1, \ldots, C_k)$. Thus we consider a component tree $T_{comp}(B_0)$ rooted at a component C_0 and whose nodes are components. If C is in this tree and $C \to (C_1, \ldots, C_k)$ then each C_i is a child of C in $T_{comp}(B_0)$. For technical reasons, C_0 is not B_0 but $\frac{5}{4}B_0$. For a component C, let $\Delta(C)$ be a disc that contains C. A leaf C in $T_{comp}(B_0)$ is said to be **terminal** if the radius of $\Delta(C)$ is $\leq \varepsilon$ and our Pellet predicate verifies that $\#_f(C) = \#_f(3C)$. We could then output $(\Delta(C), \#_f(\Delta(C))$.

While there are non-terminal components, the main loop of the algorithm will keep removing some non-terminal component C for processing. Under certain conditions we can try to apply a Newton step; if this step is successful, it will produce another component C' which becomes the sole child of C in the component tree. If it fails, we apply a bisection step on C that produces $k \geq 0$ children as above: $C \to (C_1, \ldots, C_k)$.

Our algorithms [2, 1] are numerical with arbitrary precision. They ultimately reduced to approximation by dyadic numbers (or BigFloats), i.e., elements of the ring $\mathbb{Z}[\frac{1}{2}] = \{m2^n : m, n \in \mathbb{Z}\}$. If B is a box of width $w = w(B)$ centered at $m = m(B)$, we write $\Delta(B)$ for the disc with center m but with radius $\frac{3}{4}w$. If B is exactly represented, i.e., m, w are given by dyadic numbers) then $\Delta(B)$ is also exactly represented. This ensures that the input arguments to our computational primitives are exact, even though their arithmetic is approximate. The algorithm explicitly specify the necessary precision to carry out each operation, and is therefore directly implementable.

Soft Graeffe-accelerated Pellet Test

Let us fix a polynomial $f(z) \in \mathbb{C}[z]$ of degree d. Our critical primitive is based on the classical test of Pellet (1881) [14]. For $k = 0, \ldots, d$ and disc $\Delta = \Delta(m, r) \subseteq \mathbb{C}$ with center m and radius r, the test $T_k(\Delta)$, amounts to the truth of this inequality:

$$\left|f_k(m)\right| r^k > \sum_{i \geq 0, i \neq k} \left|f_i(m)\right| r^i \tag{2}$$

where $f_i(m) = f^{(i)}(m)/i!$ is the i-th Taylor coefficient for If this test succeeds, it implies $\#_f(\Delta) = k$. But if it fails, we know nothing; this motivates [2] to introduce a "Graeffe-accelerated variant" of T_k called T_k^G in order get some useful information in case of failure. In particular, if $T_k^G(\Delta)$ fails then we know that $\#_f(\frac{4}{3}\Delta) > 0$ (i.e., there is a root is a slightly enlarged Δ). Next, we define the test $T_*(\Delta)$ to be successive testing of $T_k(\Delta)$ for $k = 0, \ldots, d$ until one of these tests succeed, whereupon the successful k value is returned. If no k is successful, return -1. We can define $T_*^G(\Delta)$ similarly for the Graeffe variants. The tests T_k^G and T_*^G so far are based on exact comparison as in (2). We "soften" exact comparison so that we can make a decision with only bounded precision. The $\texttt{IntCompare}(\widetilde{a}, \widetilde{b}, r)$ routine in Figure 2 compares two intervals $[\widetilde{a} \pm r]$ and $[\widetilde{b} \pm r]$. It returns a truth value from the set $\{\textbf{true}, \textbf{false}, \textbf{trufal}, \textbf{unresolved}\}$. These truth values have this meaning: If a and b are any in these intervals, then **true** implies $a > b$, **false** implies $a < b$, and **trufal** implies $\frac{2}{3}a < b < \frac{3}{2}a$, and **unresolved** makes no assertion about a, b. We regard the first three answers as definite, and the last as indefinite.

```
IntCompare(ã, b̃, r)
   Output: a value in {true, false, trufal, unresolved}
      Let a± ← max(0, ã ± r), b± ← max(0, b̃ ± r)
      If a⁻ > b⁺                          ◁ then a > b
         Return true
      If a⁺ < b⁻                          ◁ then a < b
         Return false
      If (2/3)a⁺ ≤ b⁻ < b⁺ ≤ (3/2)a⁻      ◁ then (2/3)a ≤ b ≤ (3/2)a
         Return trufal
      Return unresolved
```

Fig. 4. Integer Comparison

The **soft comparison** of two real numbers, denoted $a :: b$ is[8] the following procedure: for $p = 1, 2, 3, \ldots$, we compute $\widetilde{a}_p, \widetilde{b}_p$, the p-bit approximations of a, b, and call $\texttt{IntCompare}(\widetilde{a}_p, \widetilde{b}_p, 2^{-p})$. We stop as soon as the result is determinate, i.e., not **unresolved**. This process is guaranteed to terminate provided it is not the case that

[8] We view a and b as oracles that can return p-bit approximations for any desired p.

$a = b = 0$ (see [17]). If a and b are, respectively, the left and right hand sides of (2), then this iterative process is called the soft version of $T_k(\Delta)$, denoted $\widetilde{T}_k(\Delta)$. The other soft tests are obtained by a similar process. Denote the **soft** versions of T_k^G and T_*^G by $\widetilde{T}_k^G$ and $\widetilde{T}_*^G$, respectively. What we called the $P_0(B)$ (Pellet) predicate above can now be identified as $\widetilde{T}_0^G(\Delta(B))$.