Low-Memory Numerical Certification

Paul Breiding[1 [0000−0003−3747−9185]], Taylor Brysiewicz[2 [0000−0003−4272−5934]], and David K. Johnson[2 [0009−0007−9017−6825]]

[1] University of Osnabrück, Osnabrück, Lower Saxony, Germany
[2] Western University, London, Ontario, Canada

Abstract. We introduce a low-memory framework for certifying numerical solutions to polynomial systems which uses solution iterators and spatial partitioning trees to reduce memory requirements. We provide a prototypical algorithm, analyze its complexity, and demonstrate the memory reduction on a large example.

1 Introduction

Certification methods transform numerical solutions to polynomial systems into rigorously proven mathematical statements. The two most popular methods for performing such symbolic-numeric alchemy are Smale's α-theory [3, 11, 17] and the Krawczyk method [5, 12, 13]. Each enjoys $O(d)$ space and time complexity to certify d candidate solutions to a system F, and the Krawczyk method has been parallelized in [6]. However, the existing implementations require simultaneously holding all solutions in memory, a bottleneck for $d \gg 0$. Our method circumvents this memory issue by partitioning the solution space $\mathbb{C}^n$ into parts and certifying the candidates, represented by a solution iterator [4, 7], in each part.

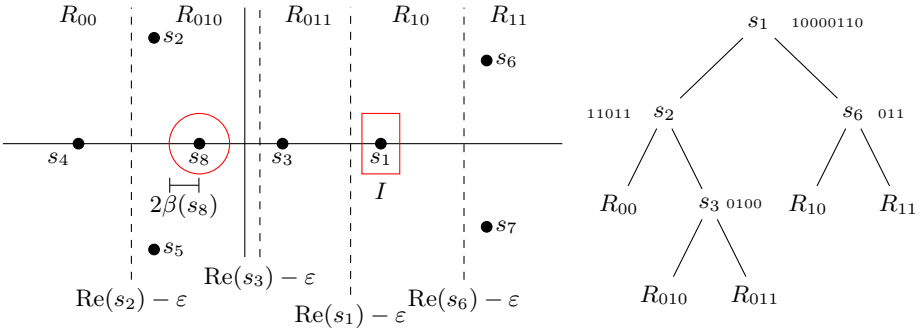

Fig. 1. (Left) A partition $\bigsqcup_{\ell \in \mathrm{leaves}(T)} R_\ell$ of $\mathbb{C}$ with candidate solutions $S = \{s_i\}_{i=1}^8$ of a real polynomial system. The red regions indicate the certified regions of the Krawczyk method and α-certification for s_1 and s_8 respectively. (Right) A binary tree encoding the partition. Each node ν represents a region R_ν, which is then refined into two smaller regions by the outcomes of $\mathrm{Re}(\mathbf{z}) < \mathrm{Re}(s_i) - \varepsilon$ for *split* s_i decorating ν. The subset of these outcomes on $R_\nu \cap S$ is cached in the corresponding bitvector.

By design, our algorithm can be applied to the datatype of a *solution iterator*. A solution iterator represents a solution set $S \subseteq \mathbb{C}^n$ by holding a single solution in memory and promising the ability to obtain the `next` solution. Introduced in [7] and expanded upon in [4], solution iterators reduce the memory complexity of many existing algorithms through their lazy-evaluation approach. In the present work, we extend the applicability of solution iterators to certification techniques.

After fixing some number $k \in \mathbb{N}$, our main idea is to partition S into parts of size no larger than k and then collect and certify each of these m parts, one at a time. Practically, this approach requires representing a partition of S in (1) a computationally efficient manner such that (2) one may certify there is no overlap between the certified regions in distinct parts. A *binary spatial partitioning* (BSP) tree, as shown in Figure 1 (right), addresses both concerns.

Algorithm 1 builds a BSP tree from a solution iterator which is, heuristically, balanced; in particular $m \approx d/k$. The user may sacrifice some storage to make this tree much more time-efficient via *bitmasks*. Algorithm 2 takes the tree along with the solution iterator and certifies all parts. Table 1 summarizes the complexity of our main algorithm, Algorithm 3, which combines these subroutines.

	Space	Time in # evaluations		
Bitmasks		Bit lookups	$<$	`next`
Krawczyk	$d\log_2(m) + 384nk$	$O(dm\log(m))$	$O(d\log(m))$	$O(d\log(m))$
α-certification	$d\log_2(m) + 384nk + 128k$	$+\, d\log(m)^2)$		
No Bitmasks		Bit lookups	$<$	`next`
Krawczyk	$64(m-1) + 384nk$	0	$O(dm\log(m))$	$O(dm)$
α-certification	$64(m-1) + 384nk + 128k$			

Table 1. Space and time complexity summary of our main algorithm, Algorithm 3.

2 Certification background

Consider a polynomial system

$$F : f_1(\mathbf{x}) = \cdots = f_n(\mathbf{x}) = 0 \quad \text{where} \quad f_1, \ldots, f_n \in \mathbb{Q}[x_1, \ldots, x_n] = \mathbb{Q}[\mathbf{x}]$$

and a set $S = \{s_1, \ldots, s_d\} \subseteq \mathbb{C}^n$ of floating point approximations of roots of F called *candidates*. There are two main algorithms which convert S into some mathematical result: Smale's α-theory and the Krawczyk method. Both methods deduce facts about the Newton operator $N_F(\mathbf{x}) = \mathbf{x} - \mathrm{Jac}_F(\mathbf{x})^{-1} \cdot F(\mathbf{x})$.

Certification by α-theory
In *Smale's α-theory*, one certifies a candidate $s \in \mathbb{C}^n$ by first computing some constants $\alpha(F,s), \beta(F,s)$ and $\gamma(F,s)$ (see [11] for details). The foundational result comes from [3, Page 160].

Theorem 1. *If $\alpha(F,s) < \frac{13-3\sqrt{17}}{4} \approx 0.157671$ then s converges quadratically to a true root z of F under repeated application of Newton's method. Moreover, the true root z is at most $2\beta(F,s)$ from s.*

Practically, one either uses a rational approximation of s to compute exactly, or interval arithmetic methods [14].

Interval certification
The second certification paradigm is *the Krawczyk method* [12] which shows that the Newton operator is a contraction on an interval box $I \subseteq \mathbb{C}^n \cong \mathbb{R}^{2n}$ containing s. It verifies this using interval arithmetic (see [5] for details). Successfully showing that N_F is a contraction on I proves that I contains a unique root by Banach's fixed point theorem [1]. For details on the Krawczyk method, see [15].

Theorem 2. *Let $I \subset \mathbb{C}^n \cong \mathbb{R}^{2n}$ be an interval box. If N_F is a contraction on I then I contains a unique root of F.*

Parallel certification using distance to a reference point
Both α-certification and the Krawczyk method have three core capabilities:

1. Associate to an approximation s a true root z of F.
   (α-theory) $s \to z$ under N_F (Krawczyk) $I \to z$ under N_F
2. Provide a *certificate region* containing s and the true root z of F.
   (α-theory) $B_{2\beta(F,s)}(s)$ (Krawczyk) I
3. Check if two certified approximations s, s' correspond to distinct roots.
   (α-theory) $||s - s'|| > 2(\beta(F,s) + \beta(F,s'))$ (Krawczyk) $I \cap I' = \emptyset$

Certifying a single point involves Item 1 in the above list. The difficulty of parallel or low-memory certification, however, is the need to show that the true roots obtained from Item 1 are distinct. Doing so involves Item 3 which, *prima facie*, requires $O(d^2)$ comparisons and obstructs trivial parallelizability. Alternatively, one can compute (in parallel) the distance intervals that the certificate regions, obtained in Item 2, inhabit with respect to some fixed reference point $\mathbf{z} \in \mathbb{C}^n$. Then, one must show only that the candidates whose intervals intersect are distinct, which may be done in parallel. This technique is implemented in [6]. However, incorporating it into the lazy-evaluation framework of solution iterators still requires significant memory costs (see Example 1).

3 Binary spatial partitioning trees

Other than solution iterators, the main datatype involved in our algorithms is that of a *binary spatial partitioning* (BSP) tree. Our version of a BSP tree is simplistic, as we do not require nor benefit from more intricate constructions, such as kd-trees [9, Chapter 5]. We seek only to induce a spatial partition on a set S of d solution candidates such that no part has size larger than k.

Define a partial order on $\mathbb{C}^n$ using just the real part of the first coordinate:

$$\mathbf{z} = (z_1, \ldots, z_n) < (w_1, \ldots, w_n) = \mathbf{w} \iff \mathrm{Re}(z_1) < \mathrm{Re}(w_1).$$

Note that conjugate points $\mathbf{z}$ and $\overline{\mathbf{z}}$ are incomparable. For any $\mathbf{z} \in \mathbb{C}^n$ we write

$$H_{\mathbf{z},-} = \{\mathbf{p} \in \mathbb{C}^n \mid \mathbf{p} < \mathbf{z}\} \qquad H_{\mathbf{z},+} = \{\mathbf{p} \in \mathbb{C}^n \mid \mathbf{p} \not< \mathbf{z}\}$$

for the left open halfspace and right closed halfspace defined by $\mathbf{z}$, which is called the *splitting point* or *split*. In practice, we shift the split by some tolerance $\varepsilon > 0$ and define $\mathcal{H}_{\mathbf{z},-} = H_{\mathbf{z}-\varepsilon\cdot e_1,-}$ and $\mathcal{H}_{\mathbf{z},+} = H_{\mathbf{z}-\varepsilon\cdot e_1,+}$. This is necessary since these halfspaces must be later shown (step 4 of Algorithm 2) to contain certificate regions of split points, as shown in the left of Figure 1. In practice, ε must be chosen so that no two real first coordinates differ by precisely ε. Ideally, ε is sufficiently small so that $\mathcal{H}_{\mathbf{z},+} = H_{\mathbf{z},+}$ since otherwise, solutions with distinct real first coordinates may become numerically incomparable via $\mathcal{H}_{\mathbf{z},+}$.

Let T be a binary tree whose internal nodes represent splits in $\mathbb{C}^n$ so that labels of left (resp. right) descendants of a split $\mathbf{z}$ belong to $\mathcal{H}_{\mathbf{z},-}$ (resp. $\mathcal{H}_{\mathbf{z},+}$). Every node ν, including the leaves, of such a tree thus represents a region obtained via the intersection of the halfspaces indexed by its split ancestors:

$$R_\nu := \left(\bigcap_{\nu \prec \mathbf{z}} \mathcal{H}_{\mathbf{z},-}\right) \cap \left(\bigcap_{\nu \succ \mathbf{z}} \mathcal{H}_{\mathbf{z},+}\right).$$

Here $\prec$ indicates "left descendant" and $\succ$ indicates "right descendant". Such a tree is called a *binary spatial partitioning* (BSP) tree since the leaf regions of T partition $\mathbb{C}^n$. The following algorithmic structure builds a BSP tree iteratively.

Algorithm 1: Build BSP Tree

Input : A set $S \subseteq \mathbb{C}^n$ of size d
A number $k \in \mathbb{N}$ bounding the size of each part
Output: A BSP tree T whose leaf regions each contain at most k points of S

1 Choose $\mathbf{z}_1 \in \mathbb{C}^n$.
2 Initialize T with root (depth 0) labeled with the split $\mathbf{z}_1$
(implicitly branching children representing $\mathcal{H}_{\mathbf{z}_1,-}$ and $\mathcal{H}_{\mathbf{z}_1,+}$).
3 **while** *there exists a leaf* ℓ *of* T *such that* $|R_\ell \cap S| > k$ **do**
4 Choose $\mathbf{z} \in R_\ell$ and assign $\mathbf{z}$ to ℓ (implicitly branching children).
5 **return** T

Algorithm 1 terminates if no more than k points in S are incomparable. In practice, k may be chosen to be quite large, and one may adapt $<$ to break ties. Steps 1 and 4 require a method for choosing splits. Choices include

1. (Median) Choose the median of the real part of the first coordinate of $R_\nu \cap S$.
2. (Mean) Choose the mean of the real part of the first coordinate of $R_\nu \cap S$.
3. (Random) Choose a uniform random point in $R_\nu \cap S$.

The *Median* method is used in many algorithms [9] as it guarantees that the children represent subsets of S of almost equal sizes at each step, inducing a tree which is *balanced*. The *Mean* method will produce an approximately balanced tree provided the points S are spatially well-behaved. Similarly, for large d, the *Random* method will produce an approximately balanced tree. For ease of analysis, we henceforth assume that any of these methods produces a balanced BSP tree. In particular, we assume that there are $m \approx d/k$ leaves, each indexing a region with $\approx k$ points of S. Thus, the tree has height $\log_2(m)$ and computing the leaf region containing some $\mathbf{z} \in \mathbb{C}^n$ costs $\log_2(m)$ evaluations of $<$.

4 Implementing BSP trees over iterators

A *solution iterator* $\mathcal{I}$ is a data structure which represents the d solutions $S \subseteq \mathbb{C}^n$ to a polynomial system. It holds a single solution s_i in memory and provides the user with access to a `next` function that exchanges the solution s_i in memory with s_{i+1}. In [4] and [7], the `next` function requires some amount of numerical *path-tracking*, the core algorithm of numerical algebraic geometry [10]. Path-tracking is much more expensive than accessing a point in a list.

Algorithm 1 can be performed using an iterator $\mathcal{I}$ for S as input, instead of the solution set S itself. Storing the real part of the first coordinates of the splits on each of the $m-1$ internal nodes of the tree allows one to query which leaf region contains a given point. Finding the elements $R_\nu \cap S$ in a region at depth δ in step 3 requires d `next` calls and $d\delta$ evaluations of $<$. Doing this for all $2m-2$ non-root regions requires $2dm-2d$ `next` calls and $\sum_{\delta=1}^{\log_2(m)} 2^\delta d\delta = 2d(m\log_2(m)-m+1)$ many $<$-evaluations.

Algorithm 1 is iterative and so many of these $<$-evaluations are recomputed several times. As such, we may choose to cache the $<$-evaluation results in a `bitmask` b_ν of length $R_\nu \cap S$ so that the coordinate $(b_\nu)_j$ agrees with the boolean value $\mathbf{z}_\nu < (R_\nu \cap S)_j$ where $\mathbf{z}_\nu$ is the split of ν. See the right of Figure 1 for an example of such a bitmasked BSP tree.

When bitmasking the BSP tree, all of the bitmasks at depth δ may be constructed using a single pass of $\mathcal{I}$. Doing so costs one `next`-call, δ-many bit lookups, and one $<$-evaluation for each of the d solutions. Summing over the $\log_2(m)$-many levels constructed in Algorithm 1 proves the following theorem.

Theorem 3. *The time and space required to construct a balanced BSP tree via Algorithm 1 is given in the following table.*

	Space	*Time*		
		Bit lookups	$<$	`next`
Bitmasked	$d\log_2(m)$	$O(d\log_2(m)^2)$	$d\log_2(m)$	$d\log_2(m)$
Not Bitmasked	$64(m-1)$	0	$O(dm\log_2(m))$	$O(dm)$

Equipped with bitmasks on each internal node of the BSP tree, one may construct an iterator $\mathcal{I}_\ell$ for each leaf ℓ whose `next` function skips over the solutions not in that leaf using the bitmasks. This can be performed without the need of several `next` calls of $\mathcal{I}$, as is the case for homotopy iterators [4].

Corollary 1. *Let T be a balanced bitmasked BSP tree. The space and time complexity to collect points in one leaf region is given in the following table.*

	Space	*Time*		
		Bit lookups	$<$	`next`
Bitmasked	$64(2n)k$	$d\log_2(m)$	0	k
Not Bitmasked	$64(2n)k$	0	$d\log_2(m)$	d

Remark 1. One may hope to improve upon the time or space complexity of our tree structure by evaluating a function which is not binary, which would

partition the solution space into many parts via one evaluation. For example, one may use a many-valued function, like partitioning the solutions into the 4^{2n} quadrants of $\mathbb{R}^{2n} \cong \mathbb{C}^n$. This saves no memory as caching the results requires moving beyond bit-vectors in such a way that precisely cancels out the memory benefits of storing fewer function caches. See Figure 2 for an example. Another drawback is that one expects to produce a less balanced tree this way.

On the other hand, this approach can save time, since the number of `next` calls is proportional to the height of the tree. One strategy is to choose splits, *a priori* unrelated to S, in hopes that they partition S into sufficiently small parts in order to bypass the iterative complexity of Algorithm 1.

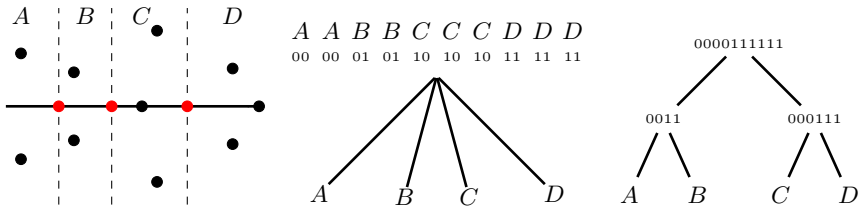

Fig. 2. (Left) 10 black points partitioned via the four regions created by three red points. (Middle) The 4-ary tree with a 4-ary vector of length 10 indicating the regions of each point, requiring $10 \cdot \log_2(4) = 20$ bits. (Right) The BSP tree obtained by partitioning via the 3 splits, in order, decorated with bit vectors also requiring 20 bits.

Remark 2. Using iterators, the *Median* method for choosing splits in Algorithm 1 seems infeasible, but the *Mean* method can be performed easily by accumulating the sum of the points in each leaf at each depth during the single iteration over $\mathcal{I}$. This is one of the observations in [7], namely that the trace of a solution set may be computed using an iterator with low-memory cost.

5 Certifying solution iterators

We arrive at the main idea for certifying solution candidates $S \subseteq \mathbb{C}^n$, to a polynomial system F, which are represented by an iterator: using a heuristically balanced BSP tree, certify the elements of S one leaf region at a time. This approach incurs the additional step of verifying that the certificate regions are contained in the interiors of the correct leaf regions. Performing this verification guarantees that the true roots associated to the candidate solutions in one region cannot coincide with the roots associated to a candidate solution in another region. For the Krawczyk method, this region is the interval box I surrounding the candidate s, and for α-certification, this region is the ball of radius $2\beta(F, s)$ centered at s. We make this precise in the following algorithm.

Algorithm 2: Certify solution iterator

Input : A solution iterator $\mathcal{I}$ for S
A balanced BSP tree T with respect to S
Output: A number of distinct certified complex and real numerical solutions

Initialize $N(\mathbb{C}) = 0$ and $N(\mathbb{R}) = 0$
for *each leaf* ℓ *in* T **do**
  Collect $R_\ell \cap S$ in memory.
  Certify the number $N_\ell(\mathbb{C})$ (resp. $N_\ell(\mathbb{R})$) of distinct complex (resp. real) candidate solutions in $R_\ell \cap S$ using α-theory or the Krawczyk method whose certificate regions are contained in R_ℓ.
  Increment $N(\mathbb{C}) += N_\ell(\mathbb{C})$ and $N(\mathbb{R}) += N_\ell(\mathbb{R})$.
return $N(\mathbb{C})$ *and* $N(\mathbb{R})$.

Theorem 4. *The space and time complexity of Algorithm 2, beyond the time cost of certifying d solutions, is given in the following table.*

	Space	*Time*		
Bitmasks		*Bit lookups*	$<$	`next`
Krawczyk	$384nk$	$dm \log_2(m)$	$2d$	d
α*-certification*	$384nk + 128k$			
No Bitmasks		*Bit lookups*	$<$	`next`
Krawczyk	$384nk$	0	$dm \log_2(m)$	dm
α*-certification*	$384nk + 128k$			

Proof. Both the Krawczyk method and α-certification (with interval arithmetic) require an interval box for k solutions upon collection. The interval box involves $2n$-many complex numbers, and the solution uses n complex numbers. Since a complex float costs 128 bits, and we are collecting at most k solutions, the space complexity follows for the Krawczyk method. For α-certification, we require another $2k$ real numbers for the k-many values of β, represented via intervals. The time complexities follow from Corollary 1, multiplied by the number m of leaves (note that $mk = d$) along with the observation that certification region checks cost two $<$-evaluations each. Each solution is certified once. The time is bounded by that of a full certification since fewer distinct solution checks are required. □

Algorithm 3: Main Algorithm

Input : A solution iterator $\mathcal{I}$ for the d candidate approximate solutions S of a square polynomial system F defined over $\mathbb{Q}$.
A maximum part size k
Output: A number of distinct certified complex and real numerical solutions

Construct a BSP tree T for $\mathcal{I}$ using Algorithm 1 with some tolerance ε.
return *Algorithm 2 called on* $(\mathcal{I}, T)$.

Theorem 5. *The cost of Algorithm 3 is given by Table 1.*

Proof. Table 1 tabulates the sum of the complexity of its two subroutines: building the tree (Algorithm 1) and certifying an iterator given a tree (Algorithm 2). Thus, it is the sum of the tables given in Theorem 3 and Theorem 4. □

Often, one has *a priori* knowledge of d and n when trying to certify a large polynomial system. A natural task is to find the $k^* = d/m^*$ which minimizes the space requirements of Algorithm 3. The next result follows from basic calculus.

Theorem 6. *The memory cost of Algorithm 3, using bitmasks and the Krawczyk method for certification, is minimized at* $m^* = 384n\ln(2)$ *to be* $d\log_2(384n\ln(2)) + d\log_2(e)$. *Without bitmasks, the memory cost is minimized at* $m^* = \sqrt{6nd}$ *to be* $384nd/\sqrt{6nd} - 64 + 64\sqrt{6nd}$.

Example 1. There are $d = 666,841,088$ quadric surfaces tangent to nine general quadrics in $\mathbb{C}^3$, each of which is identified by the $n = 10$ coefficients of a degree two polynomial in three variables (see [8, 16]). Applying our technique for the Krawczyk method with bitmasking, using parts of size no larger than $k^* \approx 250,533$, has a memory cost of ≈ 0.9952 gigabytes. One must call `next` $d\log_2(m^*) \approx 10,019,198,441$ times. Although the memory cost is only around 2.98 megabytes without bitmasking, one must call `next` approximately $dm \approx 22,111,804,214,194$ times.

Next, we approximate the memory cost of the parallel approach in Section 2. Computing an interval distance to a fixed reference point requires $2d$ many 64-bit floats, and so we compute that $\frac{2\cdot 666841088\cdot 64}{8\cdot 1024^3} \approx 9.93$ gigabytes are required.

Finally, we remark on the case in which we can predict splitting points of a balanced BSP tree for S which partitions the set S into parts of size at most k. The cost of certifying this way is dominated in time by d calls to `next` and `certify`, and uses only $384nk + 64(m-1) = \frac{3840k^2 - 64k + 42677829632}{k}$ bits of memory. This is minimized at $k^* \approx 3333$ to be approximately 3.05 megabytes. ⋄

With an iterator version of certification in hand, the difficulty in executing large certification tasks now lies in the iterator construction. The current implementation of *homotopy iterators*, as described in [4], uses the classical start systems. In the case of our example, the total degree system has too many start solutions to reliably use an iterator, and the polyhedral homotopy iterator requires a prohibitively expensive mixed volume computation. Monodromy coordinates, as described in [7], could make the problem feasible, but have yet to be implemented.

A version of iterator certification, based on the ideas in this paper, has been implemented in the `julia` [2] package `HomotopyContinuation.jl` [6] using the Krawczyk method only since α-certification is not implemented in that package.

Acknowledgements

PB was supported by DFG, German Research Foundation – Projektnummer 445466444. TB and DKJ were supported by NSERC discovery grant RGPIN-

2023-03551. The authors are grateful to Michael Burr for his suggestions regarding BSP trees and to the anonymous referees for their helpful comments which have improved this paper.

References

1. Stefan Banach. Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales. *Fundamenta Mathematicae*, 3:133–181, 1922.
2. Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.
3. Lenore Blum, Felipe Cucker, Michael Shub, and Steve Smale. *Complexity and Real Computation*. Springer-Verlag, New York, 1998. With a foreword by Richard M. Karp.
4. Paul Breiding, Taylor Brysiewicz, and Hannah Friedman. Homotopy iterators. *arXiv:2509.08084*, 2025.
5. Paul Breiding, Kemal Rose, and Sascha Timme. Certifying zeros of polynomial systems using interval arithmetic. *ACM Transactions on Mathematical Software*, 49(1):1–14, 2022.
6. Paul Breiding and Sascha Timme. HomotopyContinuation.jl: A package for homotopy continuation in Julia. In *Mathematical Software–ICMS 2018: 6th International Conference, South Bend, IN, USA, July 24-27, 2018, Proceedings 6*, page 458–465. Springer, 2018.
7. Taylor Brysiewicz. Monodromy coordinates. In Kevin Buzzard, Alicia Dickenstein, Bettina Eick, Anton Leykin, and Yue Ren, editors, *Mathematical Software – ICMS 2024*, pages 265–274, Cham, 2024. Springer Nature Switzerland.
8. Taylor Brysiewicz, Claudia Fevola, and Bernd Sturmfels. Tangent quadrics in real 3-space. *Le Matematiche*, 76(2):355–367, 2021.
9. Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, Berlin, Heidelberg, 2008.
10. Jonathan D. Hauenstein and Andrew J. Sommese. What is numerical algebraic geometry? *Journal of Symbolic Computation*, 79:499–507, 2017. SI: Numerical Algebraic Geometry.
11. Jonathan D. Hauenstein and Frank Sottile. Algorithm 921: alphaCertified: Certifying solutions to polynomial systems. *ACM Transactions on Mathematical Software*, 38(4):28:1–28:20, 2012.
12. Rolf Krawczyk. Newton-Algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken. *Computing*, 4(3):187–201, 1969.
13. Kisun Lee. Certifying approximate solutions to polynomial systems on Macaulay2. *ACM Communications in Computer Algebra*, 53(2):45–48, 2019.
14. Kisun Lee. Effective alpha theory certification using interval arithmetic: Alpha theory over regions. In Kevin Buzzard, Alicia Dickenstein, Bettina Eick, Anton Leykin, and Yue Ren, editors, *Mathematical Software – ICMS 2024*, pages 275–284, Cham, 2024. Springer Nature Switzerland.
15. Siegfried M. Rump. Solving algebraic problems with high accuracy. In Ulrich W. Kulisch and Willard L. Miranker, editors, *A New Approach to Scientific Computation*, pages 51–120. Academic Press, 1983.
16. Hermann Schubert. *Kalkül der abzählenden Geometrie*. Springer-Verlag, Berlin, 1979. Reprint of the 1879 original.

17. Steve Smale. Newton's method estimates from data at one point. In *The Merging of Disciplines: New Directions in Pure, Applied, and Computational Mathematics*, pages 185–196. Springer, New York, 1986. Proceedings of a conference held in Laramie, Wyoming, 1985.