Subdivision Algorithms for Complex Root Isolation: Empirical Comparisons

Narayan Kamath
Kellogg College
University of Oxford

A thesis submitted for the degree of
Master of Science
Trinity 2010

Abstract

This thesis deals with the application of subdivision based algorithms to the problem of isolating the roots of a complex polynomial. We provide a comprehensive comparison of the performance of three interval arithmetic based predicates (the interval Newton, Krawczyk and Hansen-Sengupta operators) with predicates based on complex analysis (the CEVAL algorithm and Yakoubsohn's approach). In addition, we include a treatment of the mathematical theory behind these operators.

Acknowledgements

I would sincerely like to thank Prof. Chee Yap and Dr. Irina Voiculescu for their insightful and patient supervision of my work through this project. I would like to thank Prof. Yap for providing me with a challenging and interesting research problem to work on, and for taking the time to help me along the way despite his numerous other commitments. In addition, I would also like to thank Dr. Irina Voiculescu for her help with various aspects of my work and for her thorough and detailed feedback on my writing. I would also like to thank Mrs. Lotti Ekert and the Numerical Analysis Group for allowing me access to books in their collections.

I am indebted to my parents and to my friends for their numerous entertaining phone calls, hours spent playing tennis, inherent willingness to join me at the pub, and for their excellent music recommendations, all of which had a substantial (positive) impact on the quality of my work.

Contents

Chapter 1

Introduction

1.1 Polynomials and their roots

Consider a univariate polynomial p with coefficients a_i in the complex field $\mathbb{C}$, for $z \in \mathbb{C}$:

$$p(z) = \sum_{i=0}^{n} a_i z^i. \tag{1.1}$$

The **fundamental theorem of algebra** then tells us that this polynomial of degree n will have exactly n complex roots[1] [43]. This gives us an alternate representation of the polynomial p in terms of its roots α_i :

$$p(z) = \sum_{i=0}^{n} a_i z^i = a_n \prod_{i=1}^{n} (z - \alpha_i). \tag{1.2}$$

The theorem however, is not constructive and does not tell us *how* to calculate these n roots, just that their existence is guaranteed. Methods of calculating polynomial roots have therefore been the subject of intense and detailed study throughout the history of mathematics.

The closed form expression[2] for the roots of a quadratic (degree $n = 2$) polynomial has been known since the middle ages, while the cubic and quartic polynomials ($n = 3, 4$) were solved comparatively later in the mid 16th century [26]. In the 19th century, the Abel-Ruffini theorem asserted that the roots of polynomials of degree $n = 5$ and above are incapable of a closed form representation. This theorem along with the advent of the modern computer shifted the focus of research in this area to methods that could be executed by a computer. A detailed and comprehensive survey of advances in root finding techniques has been compiled by Victor Pan [45], and for a general bibliography see [26].

A large fraction of these approaches fall under the category of Numerical methods i.e., methods that use approximations and iterative improvements to calculate or approximate roots. The problem in this context is often thought of as being split into two separate subproblems, **root isolation** and **root refinement**. Root isolation is the problem of calculating connected finite bounded regions of $\mathbb{C}$, each of which will contain exactly one root of the polynomial p. Root

[1] This is sometimes stated as : "Every non constant univariate polynomial whose coefficients are in the complex field $\mathbb{C}$ has at least one complex root". The form that we stated is a direct consequence of this form. Assume p is a non constant polynomial of degree $n > 1$ that has at least one complex root, say α. We can then divide p by $(z - \alpha)$ and apply the theorem to $p/(z - \alpha)$ which is also a non constant polynomial with coefficients in the complex field, and so on to count n roots until we are left with the constant polynomial

[2] A closed form representation is a representation of the roots in terms of the coefficients of the polynomial that uses a finite number of mathematical operations drawn from $\left\{+, -, \cdot, \div, \sqrt[n]{}\right\}$.

refinement on the other hand, is the problem of producing successively better approximations of the roots (often correct to a given target precision) from these isolating regions.

Our thesis deals with the problem of **isolating roots** using **subdivision methods**. A subdivision based method can be viewed as a kind of a recursive search procedure among the set of possible roots of the input polynomial. (Which is, in the general case $\mathbb{C}$.) This search is based on predicates that can either exclude parts of the search space, or include them in the algorithm output. We present a detailed study of some of these predicates in this thesis.

1.2 Aims and Motivation

1.2.1 Aims

The aim of this work is to provide an empirical comparison between closely related subdivision algorithms for isolating complex roots of a polynomial p. We compare three predicates based on interval Newton methods with a predicate based on complex analysis.

Although interval Newton methods are traditionally viewed as methods for root refinement, Moore has shown how such operators can also be used for root isolation when combined with some form of generalised bisection [27]. Volker Stahl in [38] has provided a comprehensive and detailed analysis of the application of interval arithmetic based bisection methods to the solution of systems of non linear equations. Our work is a contribution along these lines, except that we deal with the specific case of a non linear system arising out of the real and imaginary parts $u(x, y)$, $v(x, y)$ of a complex polynomial p.

Each of the three operators we consider, the interval Newton operator due to Moore [28], Krawczyk's operator [24] and Hansen and Sengupta's operator [15] are based on the Newton-Raphson iteration. Theoretically, we have an order among these operators in terms of the tightness of their output intervals. Here, we define the ordering "$>$" and say, for two operators P, Q that $P > Q$ iff $P(\mathbf{X}) \subseteq Q(\mathbf{X})$ for every interval vector $\mathbf{X}$ that they operate on, for a given system of equations. We have, due to [34] :

$$\text{Interval Newton} > \text{Hansen-Sengupta} > \text{Krawczyk}.$$

The interval Newton operator is therefore the "strongest", and we would expect it to succeed more often on a given set of inputs than the other operators. However, these three operators do not have equal computational costs. For instance, the interval Newton operator involves the inversion of an interval matrix, while the other two operators do not. Our aim is to implement all three of these operators on a common platform, and compare their **practical** performance in isolating the roots of a wide variety of polynomials.

In addition to the three interval analysis based approaches above, Yakoubsohn introduced a method to calculate ε-boxes (not necessarily isolating) around roots based on a complex analytic predicate [10]. Yakoubsohn's algorithm is based on a single "exclusion predicate" which certifies that a given box does not contain any zeros. In contrast, Sagraloff and Yap [37] introduced another algorithm for complex root isolation that has a similar exclusion predicate, but it also has an "inclusion predicate" that certifies that a given box contains exactly one zero. The algorithms in [37] were motivated by research [7, 36] in computing isotopic approximations to hypersurfaces given as the zero set of a function $f : \mathbb{R}^n \to \mathbb{R}$. Our aim is to compare the efficiency of these approaches to each other, and to the interval arithmetic based operators above.

Additionally, we aim to provide a writeup of the theory behind each of these operators and to release our implementation as a part of the free and open source Core library [9].

1.2.2 Motivation

Though it has been known (see [27, 38, 34]) that the interval arithmetic based operators above can potentially be used as inclusion tests in a subdivision algorithm, we note that there has not been an analysis of their performance in practice. Further, though the CEVAL algorithm has a well developed complexity analysis, it is lacking a practical implementation. We are therefore motivated by the need to fill this gap between theory and practice, and to look at whether the theoretically strongest predicate also offers the best practical performance.

1.3 Contributions

- We implemented subdivision algorithms based on the Newton, Krawczyk and Hansen Sengupta operators and measured their performance. Our experiments show that all three of these predicates have very similar performance, and though in theory the Krawczyk operator is the weakest test, it might be a viable practical choice because it is efficient and easy to implement.

  Further, our results show that the performance of all of these predicates degrade as the degree of the polynomial increases beyond $n = 20$. Their performance is also much slower than complex analytic predicates such as the CEVAL algorithm.

- We provide the first implementation of the CEVAL algorithm and measure its performance. Our experiments show that it is very fast and robust on polynomials of degree $n < 90$. Its speed is shown to be comparable to polynomial root finders based on competing approaches, and as it provides stronger guarantees than Yakoubsohn's exclusion approach (i.e., that its output disks are isolating) with no significant performance drawbacks, its use can be recommended.

- Though not directly related to the aims of this thesis, we contributed various improvements to the CXY implementation [25]. Our performance profiling and improvements result in the implementation running between 10 and 30 times faster than before. Our new data structures and implementation will form the basis for future extensions of this algorithm.

- As a part of our work, we contributed libraries for interval arithmetic and basic complex arithmetic operations to the Core Library. In addition, we implemented a wrapper layer over the machine precision (standard) `C++` data types to make their API consistent with those of the GMP types. We made further contributions and bug fixes to the Core Linear algebra library that we used in our implementation, such as the implementation of the Bareiss algorithm (see Chapter 10 in [43]) for determinant calculation and matrix inversion.

  In addition to this, we contributed numerous bug fixes to Core, helped transition to gcc-4.2 and made Version 2.1 linkable as a shared library on all supported platforms.

1.4 Structure of this thesis

This thesis is structured into three major sections.

Chapter 2: This chapter discusses the mathematical background behind each of these operators, and where appropriate provides a proof of their correctness. At the end of this chapter, the reader will know **why** these operators work.

Chapter 3: In this chapter, we deal with implementation of these operators in `C++`. We also provide a brief introduction to the implementation of programs that deal with numbers. At the end of this chapter, the reader will know **how** these operators have been implemented.

Chapters 4, 5: These chapters detail how the operators perform against each other, and against root isolation methods based on other approaches. At the end of these chapters, the reader will know **which** of these operators performs the best, and the limitations and advantages of using each of these approaches.

Chapter 2

Background

2.1 Subdivision based methods

Subdivision based methods are employed in the solution of a wide variety of problems across different areas of computer science. The basic idea behind them is simple. An initial region of interest is recursively subdivided and subdivisions of it are included in (or excluded from) the output set depending on whether they pass certain tests (or not). To explain this paradigm in the context of complex root finding, we start with some definitions.

2.1.1 Definitions

¶1. Basics: A complex number $z \in \mathbb{C}$ is represented by $z = x + \mathbf{i}y$ where $x, y \in \mathbb{R}$ are its real and imaginary parts respectively, and $\mathbf{i} = \sqrt{-1}$. To signify this relationship, we write $x = \texttt{Re}(z)$ and $y = \texttt{Im}(z)$. Its magnitude is represented by $|z|$ where $|z| = \sqrt{x^2 + y^2}$ and its argument $\arg z = \arctan(y/x)$ is assumed to lie between $S^1 = [0, 2\pi)$.

¶2. Box: Given $z_1, z_2 \in \mathbb{C}$ define the ordering "$\leq$" by $z_1 \leq z_2$ iff $\texttt{Re}(z_1) \leq \texttt{Re}(z_2)$ and $\texttt{Im}(z_1) \leq \texttt{Im}(z_2)$. Let the ordering "$<$" be defined in the same way, with "$<$" replacing "$\leq$" throughout. Then, for a pair of complex numbers z_a and z_b st. $z_a \leq z_b$, the **box** defined by z_a and z_b is the set of complex numbers $B = \{z : z_a \leq z < z_b\}$. It is obvious that B forms a rectangle on the complex plane. The midpoint $m(B)$ and the width $w(B)$ of the box are defined as follows:

$$\begin{aligned} m(B) &:= (z_a + z_b)/2 \\ w(B) &:= \min\{\texttt{Re}(z_b) - \texttt{Re}(z_a), \texttt{Im}(z_b) - \texttt{Im}(z_a)\} \end{aligned}$$

The width of B is chosen to be the minimum of the differences of the real and imaginary parts of z_a and z_b. In most cases we deal with square boxes and both differences are equal. Lastly, the radius $r(b)$ of a box B is defined as the radius of the circle that circumscribes B. This circle will have its centre at $m(B)$, and its radius (and that of box) is given by:

$$r(B) := \frac{1}{2}\sqrt{(\texttt{Re}(z_b) - \texttt{Re}(z_a))^2 + (\texttt{Im}(z_b) - \texttt{Im}(z_a))^2} \tag{2.1}$$

It is sometimes useful to view the definition of a box in terms of the Cartesian plane $\mathbb{R}^2$. The definition of B now becomes:

$$B = \{(x, y) : \texttt{Re}(z_a) \leq x < \texttt{Re}(z_b), \texttt{Im}(z_a) \leq y < \texttt{Im}(z_b)\} \tag{2.2}$$

A box is therefore the Cartesian product of a half-open x-axis interval $[\texttt{Re}(z_a), \texttt{Re}(z_b))$, and a half-open y-axis interval $[\texttt{Im}(z_a), \texttt{Im}(z_b))$. This definition is used by methods that use ideas of Real Analysis to operate on the box B.

¶3. Inclusion predicate ($C_{in}(B)$): An inclusion predicate is a binary valued operator on a box B that returns true **only if** B is isolating. In other words, **if** $C_{in}(B)$ holds **then** B must contain exactly one root of p.

¶4. Exclusion predicate ($C_{out}(B)$): An exclusion predicate is a binary valued operator on a box B that returns true **only if** B contains no roots of p. In other words, **if** $C_{out}(B)$ holds **then** B must contain no roots of p.

2.1.2 The generic subdivision algorithm

With these definitions in hand, we can start to define the generic subdivision algorithm. Recall that we are interested in isolating all roots of p within a given box $B_0 \subseteq \mathbb{C}$. If $C_{in}(B_0)$ holds, then B_0 is isolating and our output will simply be $\{B_0\}$. On the other hand, if $C_{out}(B_0)$ holds, then B_0 contains no roots, and we stop. If neither C_{in} nor C_{out} hold then we cannot make either decision. The strategy in this situation is to subdivide B_0 and recursively apply the inclusion and exclusion predicates on the results of the subdivision. This simple idea constitutes the algorithm ISOLATE(B_0). Note that the implicit recursion in our explanation has been unrolled into a queue (alternately a stack) Q. The correctness of the algorithm does not depend on this choice.

For an excellent background on subdivision algorithms, see [25]. Much of the notation in this section borrows from this work.

```
ISOLATE($B_0$):
     INPUT : A box $B_0 \subseteq \mathbb{C}$
     OUTPUT : A list $\mathcal{L}$ of isolating boxes contained in $B_0$
1.   $Q \leftarrow \{B_0\}$. $\mathcal{L} \leftarrow \emptyset$.
2.   While $Q$ is non-empty
3.        Remove $B$ from $Q$.
4.        If $C_{out}(B)$ holds, discard $B$.
5.        Else if $C_{in}(B)$ holds:
6.             Insert $B$ into $\mathcal{L}$.
7.        Else
8.             Subdivide $B$ and insert the subdivisions into $Q$.
```

Completeness: We observe that the algorithm above is not guaranteed to halt unless C_{out} and C_{in} are well behaved. Very roughly, the algorithm will terminate if there exists an ε st. for every box B with $w(B) \leq \varepsilon$: $C_{in}(B)$ holds **if** B is isolating, and $C_{out}(B)$ holds **if** B contains no roots. Obviously, ε depends on the distribution and separation of the roots of f, but for a given f we would like our predicates to make a decision (exclude/include) for the largest possible boxes (largest possible ε), rather than suffer further subdivision before a decision is made.

Performance: The performance of the subdivision algorithm is crucially dependant on its predicates. The exclusion predicate is meant to serve as a computationally inexpensive test that can preclude large parts of our problem domain from further consideration and allow us to concentrate our effort on regions of interest.

Subdivision schemes: Note that the subdivision process above is not clearly defined and there are multiple ways to subdivide a box B into smaller boxes. The only requirement is that B is split into disjoint boxes that are subsets of B and whose union is B. The most frequently used strategy is to split B into four equal sized boxes each of which are half the width of their parent. Other possible schemes could be to split B into two either vertically or horizontally, or to in some way use the calculations performed by C_{in} or C_{out} to determine the parameters for the subdivision. We make this definition precise as we consider each specific approach in the coming sections.

Roots at box boundaries: Our definition of the boxes on the Cartesian (or complex) plane is such that the boxes are mutually exclusive. However, in reality many operators assume boxes to be the Cartesian product of closed x and y axis intervals (especially those based on interval arithmetic, see Section 2.2.1). This leads to obvious issues with roots on the boundaries of boxes, for they might be included by both (closed) boxes that contain them, or by neither depending on the predicate design. There is no general solution to such a problem, so we deal with it specifically in the case of each of the predicates we consider (See Section 2.2.6).

Practical implementation: The practical details of implementing such algorithms are dealt with in Chapter 3, but we make a few brief remarks here for the sake of completeness. To insure against issues such as loss of precision when operating at a fixed precision level, this algorithm is often run with an additional parameter that specifies the maximum allowed extent of subdivision. This could be specified either as a lower bound on the box size, or a bound on the depth of a given box from the root of the subdivision tree. Boxes that are unresolved even after being maximally subdivided are marked as **unresolved** and are usually candidates for a further stage of processing (usually, at higher precision levels using slower extended precision arithmetic).

2.2 Interval arithmetic based predicates

We consider three closely related pairs of predicates based on interval arithmetic. As the name suggests, it is an arithmetic in which the fundamental unit is an interval. The field was pioneered by Ramon Moore to deal with issues related to computational accuracy and rounding. An excellent introduction to interval arithmetic can be found in Moore's seminal work [28]. Götz Alefeld [3] was among the first mathematicians (along with Moore) to apply this system to the solution of non linear equations. We present a brief introduction to interval arithmetic in the next section.

2.2.1 Interval arithmetic

An interval I is an ordered pair of real numbers a, b with $a \leq b$. The interval I (written $I = [a, b]$) is the set

$$I = [a, b] = \{x : a \leq x \leq b\} \tag{2.3}$$

An interval of the form $[a, a]$ is said to be **degenerate** and is equivalent to the real number a. Further, it is natural to define the midpoint $m(I)$ and the width $w(I)$ of an interval as:

$$\begin{aligned} m([a, b]) &:= (a + b)/2 \\ w([a, b]) &:= (b - a) \end{aligned}$$

An interval can be viewed as a representation of the result of a calculation along with the error associated with that result. Here $m(I)$ can be viewed as the approximate value of a calculation with an error bound $|\varepsilon| = w(I)/2$. Alternately, an interval can be viewed as a pair containing the upper and lower bound to an exact result. The set of all real intervals is often denoted by $\mathcal{I}$. Therefore, if $I \in \mathcal{I}$ then $I = [a, b]$ for some $a, b \in \mathbb{R}$ and $a \leq b$.

The standard arithmetic operations are defined on intervals in a natural way. Let $\odot$ be one one of $\{+, -, \cdot, \div\}$. Then

$$[a, b] \odot [c, d] = \{x \odot y : a \leq x \leq b, c \leq y \leq d\} \tag{2.4}$$

The definition above makes it clear that the result interval is simply the set of results of an operation, when the corresponding real number operation is applied to a pair of real numbers one taken from each of the intervals involved. The definition in (2.4) leads to expressions for these results in terms of the end points of the intervals as follows.

$$\begin{aligned}
[a, b] + [c, d] &= [a + c, b + d] \\
[a, b] - [c, d] &= [a - d, b - c] \\
[a, b] \cdot [c, d] &= [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)] \\
[a, b] \div [c, d] &= [a, b] \cdot [1/d, 1/c] \qquad (0 \notin [c, d])
\end{aligned}$$

Interval addition and multiplication is both commutative and associative, and the (degenerate) intervals $[1, 1]$ and $[0, 0]$ are identities for multiplication and addition respectively.

Distributivity: Unlike most algebraic systems, the distributive law does **not** hold for Interval arithmetic as a system. However, for any intervals I, J and K

$$I \cdot (J + K) \subseteq I \cdot J + I \cdot K \tag{2.5}$$

This property is known as **subdistributivity**. In the case that I is degenerate however, distributivity does hold. We come back to this property and some if its ramifications in the next section.

Exponentiation: Consider the obvious definition of exponentiation of an interval with an exponent $n \in \mathbb{N}$:

$$I^n = I \cdot I \cdot I..(\text{n times}).$$

Clearly, this might give us a wider result interval than is defined by (2.4). For instance, when n is even and the interval I contains zero, the result interval will contain some subset of the negative reals. However, for even n, we know that the result should be of the form $[0, a]$. We can therefore obtain tighter bounds on the result interval by augmenting our definition with special rules based on such properties. For example, when $0 \in I$ we have:

$$[a, b]^n = \begin{cases} [0, \max(a^n, b^n)] & : n = 2m, m \in \mathbb{N}, 0 \in [a, b] \\ [a^n, b^n] & : n = 2m + 1, m \in \mathbb{N}, 0 \in [a, b] \end{cases}.$$

Through the rest of this thesis, we will mention specifically when such rules are used, otherwise the standard interval arithmetic operations are assumed to apply.

Division by zeroes: In our definition of the standard arithmetic operations, we did not define the $\div$ operation in the case that the divisor contained a zero. In some cases it is useful to define this operation, though its results will be intervals of infinite length. Further, the result may be expressed as the union of two disjoint intervals rather than a single result interval. Some authors [17] refer to the system above augmented with these rules as Extended Interval arithmetic. For instance, we have the rule:

$$[a, b] \div [c, d] = [-\infty, b/d] \cup [b/c, \infty] \tag{2.6}$$

when $b < 0$, $c < 0$ and $d > 0$. These rules follow from the set theoretic definitions of these operations given by (2.4). Note that even though the result interval is of infinite length, it does provide us with useful information namely that the result is bounded away from (b/d) and (b/c). A complete description of these rules is provided in [17].

2.2.2 Interval extensions of a function

Consider a real valued function $f : \mathbb{R} \to \mathbb{R}$. We look at how interval arithmetic can be used to estimate the range of f over a finite subset of its domain. Let us assume we are interested in finding the range of $f(x)$ as x varies over the interval $[a, b]$. The range R (over $[a, b]$) is given by $R = \{f(x) : a \leq x \leq b\}$ (which we also denote by $f([a, b])$). We can then construct an **interval extension** of f, denoted by $\Box f$:

$$\Box f : \mathcal{I} \to \mathcal{I} \tag{2.7}$$

such that $f([a, b]) \subseteq \Box f([a, b])$. When f is a polynomial (or rational), an interval extension can simply be constructed by replacing the real versions of the operators $\{+, -, \cdot, \div\}$ with their interval equivalents. Given the definition of these operations (2.4), it becomes clear that such an extension will contain the range of f. Note that this extension is not unique.

Unlike real number arithmetic, interval arithmetic is not distributive (2.5), and different schemes of evaluating a polynomial that differ in the order (or number) of multiplication and addition operations might yield different results. Take for example $f(x) = x(1-x)$; we estimate its range over $[1, 2]$ in two ways:

$$\begin{aligned} \Box f(I) &= I \cdot (1 - I) = [1, 2] \cdot ([1, 1] - [1, 2]) = [1, 2] \cdot [-1, 0] = [-2, 0], \\ \Box f(I) &= I - I^2 = [1, 2] - [1, 2] \cdot [1, 2] = [1, 2] - [1, 4] = [-3, 1]. \end{aligned}$$

Note that the second method, which used the fact that $f(x) = x - x^2$ produced an interval that was wider than the first. In most cases we are interested in the extension that produces the tightest bound on the range of f.

Note that this concept is extensible to functions of the form $g : \mathbb{R}^2 \to \mathbb{R}$. The extension $\Box g(I, J)$ will operate on a pair of intervals that form a box as defined in ¶2 and return an interval that contains the range of g over that box.

2.2.3 Application to exclusion predicates

We have f, a polynomial defined on the complex numbers, whose zeroes we are interested in isolating. Replacing $z = x + \mathbf{i}y$ we can rewrite f in the form

$$f(x, y) = u(x, y) + \mathbf{i}v(x, y) \tag{2.8}$$

where both u and v are real valued functions defined on the Cartesian plane. Our problem now becomes the problem of finding the roots of the system of equations:

$$\begin{aligned} u(x, y) &= 0, \\ v(x, y) &= 0. \end{aligned}$$

Consider the interval extension $\square u(X,Y)$ where X is an x-axis interval and Y is a y-axis interval associated with a box B. We then have:

LEMMA 1. *If $0 \notin \square u(X,Y)$ then $u(x,y)$ has no zero in the box B defined by the closed intervals X, Y.*

Proof. Let $I = \square u(X,Y)$ such that $0 \notin I$. Then I must be of the form $[a,b]$ where $a \leq b < 0$ or $0 < a \leq b$. From the definition of an interval extension from Section 2.2.2, the range of u must be a subset of I. In either case this is either strictly negative or strictly positive, and will not contain a zero of u. **Q.E.D.**

Due to Lemma 1, and the fact that both u and v must be zero in order for f to be zero, we now have an **exclusion predicate** that uses interval arithmetic and it is given by:

$$0 \notin \square u(X,Y) \textbf{ or } 0 \notin \square v(X,Y). \tag{2.9}$$

Note that we mentioned earlier than an interval extension for a function f is not necessarily unique. In the context of (2.9), we would prefer an extension that produced result intervals whose widths were as low as possible.

Converse of Lemma 1: The converse of Lemma 1 is not necessarily true. Consider a function f and its interval extension $\square f$ evaluated over an interval I. Now, if the range S of f over I does not contain a 0, we have no guarantee that $\square f(I)$ will not. Since $S \subseteq \square f(I)$, $\square f(I)$ can be wider than S and may contain a zero even if S does not.

In our specific case, if interval extensions of u and v from (2.8) both contain zeroes when evaluated over a box B we cannot claim that B contains a zero of f for the same reason.

Furthermore, we are interested in isolating zeroes, and a predicate of this form cannot tell us anything about the number of zeroes in a given box. In the next few sections, we introduce methods that help us do so.

2.2.4 Newton's method and its interval form

In this section we present the Newton's method for solving the system of non linear equations:

$$\begin{aligned} u(x,y) &= 0, & (2.10)\\ v(x,y) &= 0, & (2.11) \end{aligned}$$

where $f(z) = u(x,y) + \mathbf{i}v(x,y)$. Consider the function $u(x,y)$, if u is continuous and differentiable in a neighbourhood of a point (a,b) then its (bivariate) Taylor series approximation (centred at (a,b)) to the first degree is given by:

$$u(x,y) \approx u(a,b) + (x-a)u_x(a,b) + (y-b)u_y(a,b).$$

Here u_x and u_y are the partial derivatives of u with respect to x and y respectively. We are interested in finding zeroes of u and these are values of x, y for which $u(x,y) = 0$. Since we are using an approximation of the functions entire Taylor series, we can try to estimate the zeroes of u in the vicinity of (a,b) by setting $u(x,y) = 0$ above. The equation then becomes:

$$-u(a,b) = (x-a)u_x(a,b) + (y-b)u_y(a,b) \tag{2.12}$$

Proceeding in a similar manner for v with the same initial point (a,b) and noting that the zeroes that u and v share are the zeroes of the system specified by (2.10), we get:

$$\begin{aligned} -u(a,b) &= (x-a)u_x(a,b) + (y-b)u_y(a,b),\\ -v(a,b) &= (x-a)v_x(a,b) + (y-b)v_y(a,b). \end{aligned}$$

Since this is a system of linear equations in two variables x and y, we can rewrite this in the form $A \cdot (\mathbf{x} - \mathbf{x}') = \mathbf{b}$:

$$\begin{bmatrix} u_x(a,b) & u_y(a,b) \\ v_x(a,b) & v_y(a,b) \end{bmatrix} \cdot \begin{bmatrix} (x-a) \\ (y-b) \end{bmatrix} = - \begin{bmatrix} u(a,b) \\ v(a,b) \end{bmatrix}. \tag{2.13}$$

This system of equations has a unique solution when A is invertible and this solution is given by $\mathbf{x} = \mathbf{x}' + A^{-1} \cdot \mathbf{b}$. This forms the basis for an iterative process to arrive at better refinements of the roots of (2.10) in the vicinity of (a, b). This is in essence the Newton iteration (many authors call it the Newton-Raphson iteration), and it is often written as:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - J(\mathbf{x}_k)^{-1} \cdot \mathbf{f}(\mathbf{x}_k). \tag{2.14}$$

This is a rewrite of (2.13) generalised to a system of n equations in n variables where $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$, $\mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_n(\mathbf{x}))^T$, each $f_i : \mathbb{R}^n \to \mathbb{R}$ and $J(\mathbf{x})$ is the Jacobian of the system given by:

$$J(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1(\mathbf{x})}{\partial x_1} & \frac{\partial f_1(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial f_1(\mathbf{x})}{\partial x_n} \\ \frac{\partial f_2(\mathbf{x})}{\partial x_1} & \frac{\partial f_2(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial f_2(\mathbf{x})}{\partial x_n} \\ \cdots & & & \\ \frac{\partial f_n(\mathbf{x})}{\partial x_1} & \frac{\partial f_n(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial f_n(\mathbf{x})}{\partial x_n} \end{bmatrix}. \tag{2.15}$$

The conditions for convergence and the order of convergence of this iteration to a root of $\mathbf{f}$ are beyond the scope of this thesis, and can be found in Section 5.3 of [39]. Furthermore, we are not concerned about the convergence of this iteration because we do not use it as is. We will instead exploit some of its properties to construct a predicate that tests whether a box is isolating or not.

The interval analogue of this operator is defined in the natural way. The first justification for its use was provided by Moore [28]. Alefeld [2] and Hansen [16] independently came up with a globally convergent version of this operator using extended interval arithmetic. (See [28] or [34] for a detailed analysis of the same.) We stop at providing a brief description of the interval analogue of this operator before turning our attention to its use as an inclusion test.

Let $\mathbf{X} = (X_1, X_2 \ldots X_n)$ be a vector of intervals and the interval extension of the Jacobian $\square J(\mathbf{X})$ be defined using the interval extensions of each of the partial derivatives. Then, $\square J(\mathbf{X})$ will contain all $J(\mathbf{x})$ for $\mathbf{x} \in \mathbf{X}$. Finally, let the vector of midpoints of each of the intervals in $\mathbf{X}$ be denoted by $\mathbf{x_c}$. Then

$$N(\mathbf{X}) = \mathbf{x}_c - \square J(\mathbf{x})^{-1} \cdot \mathbf{f}(\mathbf{x}_c), \tag{2.16}$$

and the iteration is of the form

$$\mathbf{X}_{n+1} \leftarrow \mathbf{X}_n \cap N(\mathbf{X}_n). \tag{2.17}$$

The motivation for this type of iteration is easier to understand if we consider the univariate case where the iteration becomes

$$N(X) = m(X) - \frac{f(m(X))}{\square f'(X)}. \tag{2.18}$$

We prove the following lemma due to Moore (Lemma 7.2 in [28]) along the same lines. The proof is included here to provide an insight into why the iteration defined above might help us.

Lemma 2. *If $N(X)$ is defined by (2.18) for an interval X that contains a root x_0 of $f(x)$ of multiplicity 1, and f and f' are both continuous over the interval X, then $x_0 \in N(X)$.*

Proof. Let $x_0 \in X$ be a root of $f(x)$. If the root lies at the mid point $m(X)$ of X i.e, $x_0 = m(X)$, then $f(m(X)) = f(x_0) = 0$ and $N(X) = [x_0, x_0]$ which proves the lemma for this case.

Let us now assume that $x_0 > m(X)$, and consider the interval $[m(X), x_0] \subset X$. Given the continuity of f and f' and using the Mean Value Theorem, we have at some point $y \in [m(X), x_0]$:

$$f'(y) = \frac{f(x_0) - f(m(X))}{(x_0 - m(X))},$$

and since $f(x_0) = 0$ we get:

$$x_0 = m(X) - \frac{f(m(X))}{f'(y)}.$$

Given the properties of $\Box f'(X)$ we require $f'(y) \in \Box f'(X)$ and therefore $x_0 \in N(X)$. The case where $x_0 < m(X)$ can be proved in a similar manner by the application of the Mean Value Theorem to the interval $[x_0, m(X)]$. **Q.E.D.**

The above proof gives us a rough justification of why the interval iteration defined by (2.16) makes sense. In fact, with some stronger assumptions the rate of convergence of $N(X)$ to a root $x \in X$ can be shown to be quadratic. (See for example [28])

Note that in its current form, the Newton iteration operates on an input interval or vector of intervals (in the n dimensional case), and starting with an initial approximation converges under certain conditions to a root in the neighbourhood of the first approximation. To use it as an inclusion predicate, we require it to answer questions about the **existence** of a root in a given box B, rather than trying to refine it based on an approximation. The next section deals with how (2.16) can be used as an inclusion predicate.

2.2.5 The interval Newton operator as an inclusion test

We begin this section by making some statements about the Jacobian $\Box J(\mathbf{X})$. It is a matrix whose every element is an interval, and which contains every $J(\mathbf{x})$ as $\mathbf{x}$ varies over $\mathbf{X}$. We say $\Box J(\mathbf{X})$ is invertible if it does not contain a (real) singular matrix.

Lemma 3. *If $\Box J(\mathbf{X})$ is invertible. and for any $\mathbf{x}, \mathbf{y} \in \mathbf{X}$, if $\mathbf{f}(\mathbf{x}) = \mathbf{f}(\mathbf{y})$ then $\mathbf{x} = \mathbf{y}$.*

Proof. Recall that $\mathbf{f}$ is a vector of functions $(f_1, f_2 \dots f_n)^T$. Consider each of these functions f_i. Define a function on the scalar t by

$$g_i(t) = f_i(t\mathbf{y} + (1-t)\mathbf{x}),$$

such that $g_i(0) = f_i(\mathbf{x})$ and $g_i(1) = f_i(\mathbf{y})$. Now, g_i is continuous and differentiable as t varies across the interval $[0, 1]$ and by the Mean Value Theorem we must have:

$$f_i(\mathbf{y}) - f_i(\mathbf{x}) = g_i'(c_i)(1-0),$$

for some $c_i \in [0,1]$. However[1], $g_i'(c_i) = \nabla f_i((1-c_i)\mathbf{x} + c_i\mathbf{y}) \cdot (\mathbf{y}-\mathbf{x})$ and $f_i(\mathbf{x}) = f_i(\mathbf{y})$ since $\mathbf{f}(\mathbf{x}) = \mathbf{f}(\mathbf{y})$. This gives us:

$$0 = \nabla f_i((1-c_i)\mathbf{x} + c_i\mathbf{y})(\mathbf{y}-\mathbf{x}).$$

Proceeding componentwise in the same manner, we have similar equations for each f_i. Recognising that ∇f_i corresponds to the i^{th} row in the Jacobian $J(\mathbf{x})$ in (2.15) , we introduce the notation $J_m(\mathbf{c})$ for the Jacobian matrix obtained by evaluating the i^{th} row of partial derivatives at c_i. We use the subscript m to signify that this Jacobian is associated with the application of the Mean Value Theorem to a t interval associated with $\mathbf{x}$, $\mathbf{y}$. We also write this as $J_m(\mathbf{x},\mathbf{y})$ to strengthen this association. The i componentwise equations together give us:

$$0 = f(\mathbf{y}) - f(\mathbf{x}) = J_m(\mathbf{x},\mathbf{y}) \cdot (\mathbf{y}-\mathbf{x}). \tag{2.19}$$

Now, each $c_i \in [0,1]$ and as a consequence $J_m(\mathbf{x},\mathbf{y}) \in \square J(\mathbf{X})$ and since we know $\square J(\mathbf{X})$ does not contain a singular matrix, we must have $\mathbf{y} = \mathbf{x}$. **Q.E.D.**

An important thing that this lemma tells us is that if there exists a root $\mathbf{x} \in \mathbf{X}$ i.e., $\mathbf{f}(\mathbf{x}) = 0$ then that root is necessarily unique. We now prove the central result of this section, one that allows us to use the Newton operator $N(\mathbf{X})$ of (2.16) as an existence test. To prove this, we use a version of the Brouwer Fixed Point Theorem, which is stated below without proof. For details see Sections 54,55 of Munkres' excellent topology text [33].

Theorem 4 (Brouwer Fixed-Point Theorem). *Every continuous function f from a convex compact subset S of a Euclidean space to S itself has a fixed point.*

We now state and prove the main result, which is due to Moore [28] (for part 2) and Nickel [35] for the main result.

Theorem 5. *Let $\mathbf{X}$ be a vector of intervals and $\mathbf{f}$ a vector of functions that are continuous and differentiable over $\mathbf{X}$. Further, let $N(\mathbf{X})$ be defined by (2.16). Then:*
(1) If $N(\mathbf{X}) \subseteq \mathbf{X}$, then $\mathbf{f}(\mathbf{x}) = 0$ has a unique root in $\mathbf{X}$ i.e., $\mathbf{X}$ is isolating.
(2) If $N(X_i) \cap X_i = \emptyset$ for any X_i, a component of $\mathbf{X}$, then $\mathbf{f}(\mathbf{x}) = 0$ has no roots in $\mathbf{X}$.

Proof. Consider the mapping $P(\mathbf{x}) = \mathbf{x}_c - J_m(\mathbf{x},\mathbf{x}_c)^{-1}\mathbf{f}(\mathbf{x}_c)$, where $\mathbf{x},\mathbf{x}_c \in \mathbf{X}$ and $\mathbf{x}_c$ is the centre of $\mathbf{X}$ as defined in (2.16). From the proof of Lemma 3, $J_m(\mathbf{x},\mathbf{x}_c) \in \square J(\mathbf{X})$ and as a consequence $P(\mathbf{x}) \in N(\mathbf{X})$ for all $\mathbf{x} \in \mathbf{X}$. Therefore, the range of P, $P(\mathbf{X}) \subseteq N(\mathbf{X})$. We now have $P(\mathbf{X}) \subseteq N(\mathbf{X}) \subseteq \mathbf{X}$.

Also, $\mathbf{X}$ is of the form $\prod^{n} [a_i, b_i]$ and is a closed subset of the n-dimensional Euclidean space. It is also bounded, since the intervals involved are finite and therefore[2] it is compact.

[1]Note that here "$\cdot$" is the scalar product of two vectors. This can be derived as follows. The differentiation here $g_k'(t)$ is with respect to t and we use the chain rule.

$$\begin{aligned}
\frac{dg_k}{dt} &= \sum_i \frac{\partial f_k}{\partial x_i}\frac{dx_i}{dt} \\
&= \sum_i \frac{\partial f_k}{\partial x_i}\frac{d}{dt}(ty_i + (1-t)x_i) \\
&= \sum_i \frac{\partial f_k}{\partial x_i}(y_i - x_i) \\
&= \nabla f_k(t\mathbf{x} + (1-t)\mathbf{y}) \cdot (\mathbf{y}-\mathbf{x})
\end{aligned}$$

[2]See Section 27 of [33], in particular Theorem 27.3.

By the Brouwer Fixed Point Theorem, we have P, a mapping from a closed compact subset of a Euclidean space to itself and it must necessarily have a fixed point, say $\mathbf{x}^*$ with $P(\mathbf{x}^*) = \mathbf{x}^*$ and:

$$\mathbf{x}^* = \mathbf{x}_c - J_m(\mathbf{x}^*, \mathbf{x}_c)^{-1}\mathbf{f}(\mathbf{x}_c).$$

This can be rewritten into:

$$J_m(\mathbf{x}^*, \mathbf{x}_c) \cdot (\mathbf{x}^* - \mathbf{x}_c) = -\mathbf{f}(\mathbf{x}_c). \tag{2.20}$$

This is the form of (2.19) and hence, the LHS of (2.20) above is equal to $\mathbf{f}(\mathbf{x}^*) - \mathbf{f}(\mathbf{x}_c)$. Therefore:

$$\begin{aligned} -\mathbf{f}(\mathbf{x}_c) &= J_m(\mathbf{x}^*, \mathbf{x}_c) \cdot (\mathbf{x}^* - \mathbf{x}_c) \\ &= \mathbf{f}(\mathbf{x}^*) - \mathbf{f}(\mathbf{x}_c) \end{aligned}$$

and this implies $\mathbf{f}(\mathbf{x}^*) = 0$. The uniqueness of this zero is guaranteed by Lemma 3.

As for the second part of the theorem, observe from our proof above that the roots of $\mathbf{f}$ are fixed points of $P(\mathbf{x})$ as defined above. If the range of $P(\mathbf{x})$ does not intersect $\mathbf{X}$, then P cannot have any fixed points in $\mathbf{X}$ and as a result neither can $\mathbf{f}$ have any roots in $\mathbf{X}$. **Q.E.D.**

2.2.6 The interval Newton operator and the subdivision algorithm

The results above have been proved for the general n dimensional case, and are applicable to our system of equations (2.10) which is of dimension $n = 2$. Recall the definition of a box from ¶2 is expressed in terms of two half-open intervals defined in (2.2). However, all of the theory above was described in terms of closed intervals. To apply these operators, we will need to treat boxes as the Cartesian products of closed x and y axis intervals, say $[x_1, x_2] \times [y_1, y_2]$.

Roots at boundaries: This change brings with it the problem of roots that lie on the (shared) boundary of two boxes. It might be that our inclusion predicate classifies both boxes as isolating the root that lies on their shared boundary. Even worse is the case of the root that lies on the shared corner of four boxes. Subdividing boxes will not make the problem go away, because the root will continue to be shared by some children of subdivided boxes. This is a problem that is generally glossed over by existing literature in this area, and there are two general approaches that we can take to solving it. Both of these approaches can be viewed as some sort of "fudge factors" to deal with this corner case. However, we emphasise that despite their use the end results are *always* isolating boxes.

- The first approach could be to subdivide the region to the maximal extent possible, and then coalesce unresolved boxes (these will include boxes that share a root on their boundary). Coalescing is performed by grouping together unresolved boxes that share a common edge, and by constructing a minimal bounding box around these groups. We can then run our subdivision algorithm with these bounding boxes as starting regions to isolate roots that they are suspected to contain.

- The second approach could be to perturb boxes during the subdivision process. If we suspect that a box shares a root on its boundary with another, during its next subdivision we adjust its boundaries outward by a fixed ε, while at the same time adjusting the boundaries of the corresponding neighbours inwards by the same amount.

Of the two, we choose the first approach due to the simplicity of its implementation. Our last observation is that the number of boxes for which this will happen is bounded by $4n$ where

n is the degree of the polynomial whose roots we are isolating. We provide examples of this situation in Section 3.4.4.

It is crucial to note that neither of these approaches guarantees termination. In the first case, our algorithm run on the coalesced boxes could itself complete with unresolved boxes. Further, we might duplicate isolating regions if bounding boxes intersect. In the second case, it is possible (though with a very low probability) that roots will continue to be at box boundaries despite the perturbation.

In his thesis [38, Section 5.9.1, Algorithm 5.9.4], Stahl proposes a method to overcome this issue of roots that lie on box boundaries. The main idea behind his approach is to dilate every box B to obtain a dilated box $\hat{B}$. The operators are then applied on this box until one of two things happens:

- $N(\hat{B}) \cap B = \emptyset$ in which case B will contain no roots of f.
- $N(\hat{B}) \subseteq \hat{B}$ which guarantees that $\hat{B}$ is isolating. However, $N(\hat{B})$ need not be a subset of B. This appears to leave open the issue of two isolating boxes being generated around the same root.

Setting aside the problem of roots on boundaries for now, we have a predicate for determining if a box B is isolating:

```
1.  If $N(B) \cap B = \emptyset$
2.      Discard $B$
3.  Else If $N(B) \subseteq B$
4.      Insert $B$ into the output.
5.  Else
8.      Subdivide $B$
```

With this complete, we move on to our next inclusion predicate.

2.2.7 The Krawczyk operator

The Newton method introduced above requires the inversion of an interval valued matrix, and can therefore be relatively expensive to evaluate. Krawczyk provides an alternate formulation that is almost equivalent to the Newton operator, but does not necessarily require a matrix inversion. The **Krawczyk operator** is given by:

$$K(\mathbf{X}) := \mathbf{y} - Y \cdot \mathbf{f}(\mathbf{y}) + \{I - Y \cdot \square J(\mathbf{X})\}(\mathbf{X} - \mathbf{y}), \tag{2.21}$$

where $\mathbf{y} \in \mathbf{X}$, Y is any nonsingular real matrix, and I the identity matrix. Here, $\mathbf{y}$ is typically taken to be the mid point of $\mathbf{X}$. Observe that that the term $\mathbf{y} - Y \cdot \mathbf{f}(\mathbf{y})$ does not involve any intervals, and is very similar to the Newton operator with $Y \approx J(\mathbf{y})^{-1}$. The second half of the expression involves non degenerate intervals i.e., $Y \cdot \square J(\mathbf{X})$ and $\mathbf{X}$ itself. This can be viewed as an error estimate on $\mathbf{y} - Y \cdot \mathbf{f}(\mathbf{y})$ due to the use of Y as an approximation to J^{-1}. Observe that as Y becomes a better approximation of $J(\mathbf{y})^{-1}$ the term $\{I - Y \cdot \square J(\mathbf{X})\}$ will get closer to zero making the effects of the associated term negligible.

Krawczyk's operator leads to the following analogue of the Newton iteration presented in the previous section:

$$\mathbf{X}_{(k+1)} \leftarrow \mathbf{X}_{(k)} \cap K(\mathbf{X}_{(k)}), \qquad k = 0, 1, 2, \ldots.$$

We will show that if $\mathbf{X}$ contains a solution to $\mathbf{f}(\mathbf{x}) = 0$, then so does $K(\mathbf{X})$. We will also discuss an inclusion test based on this operator which is very similar to that of the previous section. This is due to the work of Moore who provided a proof in [29].

Although the operator $K(\mathbf{X})$ looks quite complex, it has a straightforward underlying function:

$$P(\mathbf{x}) := \mathbf{x} - Y \cdot \mathbf{f}(\mathbf{x}), \tag{2.22}$$

where Y is any nonsingular real matrix. This is a relaxed form of the Newton iteration defined in (2.14) which requires $Y = J(\mathbf{x})^{-1}$. We start by proving a property of $P(\mathbf{x})$.

Lemma 6. *Let $\mathbf{X} \subseteq \mathbb{R}^n$ be a closed compact convex set. If $P(\mathbf{X}) \subseteq \mathbf{X}$ then $\mathbf{f}(\mathbf{x})$ has a zero in $P(\mathbf{X})$.*

Proof. Since $\mathbf{f}$ is continuous (as each f_i is), P is continuous and maps $\mathbf{X}$ into itself. From the Brouwer fixed point theorem, P has a fixed point $\mathbf{x}^* \in \mathbf{X}$ such that:

$$\mathbf{x}^* = \mathbf{x}^* - Y \cdot \mathbf{f}(\mathbf{x}^*)$$

Knowing that Y is invertible, multiplying both sides of the equation by Y^{-1} gives us $\mathbf{f}(\mathbf{x}^*) = 0$. Moreover, $\mathbf{x}^* \in P(\mathbf{X})$. **Q.E.D.**

Lemma 7. *Let $\mathbf{X}$ be convex. Then $\mathbf{f}(\mathbf{x}) - \mathbf{f}(\mathbf{y}) \in \square J(\mathbf{X})(\mathbf{x} - \mathbf{y})$ for all $\mathbf{x}, \mathbf{y} \in \mathbf{X}$.*

Proof. This is a direct consequence of (2.19) from Lemma 3. **Q.E.D.**

We now look at how $P(\mathbf{x})$ is connected to $K(\mathbf{X})$. This was Krawczyk's intuition in his initial work [24]. From the definition of P in Lemma 6, we have:

$$\begin{aligned} P(\mathbf{x}) &= \mathbf{x} - Y\mathbf{f}(\mathbf{x}) \\ &= \mathbf{x} - Y\mathbf{f}(\mathbf{x}) + (\mathbf{y} - Y\mathbf{f}(\mathbf{y})) - (\mathbf{y} - Y\mathbf{f}(\mathbf{y})) \\ &= \mathbf{y} - Y\mathbf{f}(\mathbf{y}) + \mathbf{x} - \mathbf{y} - Y[\mathbf{f}(\mathbf{x}) - \mathbf{f}(\mathbf{y})]. \end{aligned}$$

This is $P(\mathbf{x})$ rewritten in terms of some fixed $\mathbf{y} \in \mathbf{X}$. This rewritten form holds for all $\mathbf{x} \in \mathbf{X}$, and we can estimate the range of $P(\mathbf{x})$ using its interval extension (and in turn that of f). Recall that the range of P will be a subset of the interval extension $\square P(\mathbf{X})$:

$$\begin{aligned} \square P(\mathbf{X}) &= \mathbf{y} - Y\mathbf{f}(\mathbf{y}) + \mathbf{X} - \mathbf{y} - Y[\mathbf{f}(\mathbf{X}) - \mathbf{f}(\mathbf{y})] \\ &\subseteq \mathbf{y} - Y\mathbf{f}(\mathbf{y}) + [\mathbf{X} - \mathbf{y}] - Y(\square J(\mathbf{X})[\mathbf{X} - \mathbf{y}]) \quad \text{(by Lemma 7)} \\ &= \mathbf{y} - Y\mathbf{f}(\mathbf{y}) + \{I - Y\square J(\mathbf{X})\}\,[\mathbf{X} - \mathbf{y}] \\ &= K(\mathbf{X}). \end{aligned}$$

We now turn our attention to how $K(\mathbf{X})$ can be used as an inclusion predicate. The original work of Krawczyk spoke about the applicability of his operator to the problem of refining roots, rather than as an existence test. Its utility as an inclusion predicate was observed by Moore, and is a theorem from his work [29]. Note that we state a stronger version than Moore (we state that any solution of $\mathbf{f}(\mathbf{x}) = 0$ in $\mathbf{X}$ is actually in $K(\mathbf{X})$).

Theorem 8. *Let $\mathbf{y}$ be a fixed real vector in the interval vector $\mathbf{X}$, and let $K(\mathbf{X})$ be defined as in (2.21). Then:*
(1) $K(\mathbf{X}) \subseteq \mathbf{X}$ implies the existence of a zero of $\mathbf{f}$ in $K(\mathbf{X})$.
(2) If $K(\mathbf{X}) \subset \mathbf{X}$ then this zero is unique.
(3) If $K(\mathbf{X}) \cap \mathbf{X} = \emptyset$, then $\mathbf{X}$ has no zero of $\mathbf{f}$.

Proof. We have $P(\mathbf{X}) \subseteq K(\mathbf{X})$ (from our justification). This means that $K(\mathbf{X}) \subseteq \mathbf{X}$ implies $P(\mathbf{X}) \subseteq \mathbf{X}$. By Lemma 6, this implies $\mathbf{f}$ has a zero in $P(\mathbf{X})$, and hence in $K(\mathbf{X})$.

The proof for the second part is slightly more complex. In the corresponding proof for the Newton's method, we exploited the fact that the Jacobian did not contain a singular matrix. This is not necessarily the case for the Krawczyk operator.

The complete proof for this part of the theorem can be found in Lemma 3, Theorem 2 in [29]. The basic idea behind Moore's proof is that when $K(\mathbf{X}) \subset \mathbf{X}$, the sequence of intervals obtained by applying $K(\mathbf{X})$ recursively to $\mathbf{X}$ converges to a *single* point, and that point will be a root of $\mathbf{f}$.

The proof for the last part of this theorem is along the same lines as the Newton operator in Theorem 5. **Q.E.D.**

The predicate based on the Krawczyk operator takes the same form as that with the Newton operator which is:

```
1.  If $K(B) \cap B = \emptyset$
2.      Discard $B$
3.  Else If $K(B) \subset B$
4.      Insert $B$ into the output.
5.  Else
8.      Subdivide $B$
```

We now move on to the last of our inclusion tests, which is the Hansen-Sengupta operator.

2.2.8 The Hansen-Sengupta operator

We begin this section by recalling (2.19) in which we showed for any $\mathbf{x}, \mathbf{y}$ in an interval vector $\mathbf{X}$ that:

$$f(\mathbf{y}) - f(\mathbf{x}) = J_m(\mathbf{x}, \mathbf{y}) \cdot (\mathbf{y} - \mathbf{x}), \tag{2.23}$$

and that $J_m(\mathbf{x}, \mathbf{y}) \in \Box J(\mathbf{X})$. We are interested in finding those $\mathbf{y}$ for which $\mathbf{f}(\mathbf{y}) = 0$. Using this in the above equation, for a fixed $\mathbf{x_c}$ (usually the mid point of $\mathbf{X}$) we have:

$$J_m(\mathbf{x_c}, \mathbf{y})(\mathbf{y} - \mathbf{x_c}) + \mathbf{f}(\mathbf{x_c}) = 0.$$

Finding the set of solutions of this equation is as hard as the original problem itself, but what we can do is compute a superset of this solution set by replacing $J_m(\mathbf{x_c}, \mathbf{y})$ with $\Box J(\mathbf{X})$. The equation now becomes:

$$\Box J(\mathbf{X})(\mathbf{y} - \mathbf{x_c}) + \mathbf{f}(\mathbf{x_c}) = 0. \tag{2.24}$$

This is now a linear equation in $\mathbf{y}$ with interval coefficients since the elements of $\Box J$ are intervals. Hansen and Sengupta in [17] observed that the Krawczyk operator tries to solve (2.24) only approximately, and does not in general give sharp bounds on its solutions. In addition, they proposed a new operator which produces sharper bounds on the solution. This is the subject of the rest of our discussion.

Hansen observed in [15] that the solution of equation (2.24) could proceed along the lines of the Gauss-Seidel elimination, as performed on a system with real coefficients. With Smith, in [18] he also showed that sharper bounds on its solution in the interval case could be obtained by premultiplying (or conditioning) (2.24) with a suitable matrix. This is usually chosen to be the inverse of $J_c(\mathbf{X})$, the (non interval) mid point of $\Box J(\mathbf{X})$. If this premultiplier is B say, the equation then becomes:

$$A \cdot (\mathbf{y} - \mathbf{x_c}) = \mathbf{l},$$

where $A = B \cdot \square J(\mathbf{X})$ and $\mathbf{l} = -B \cdot \mathbf{f}(\mathbf{x_c})$. This remains a linear equation in $\mathbf{y}$.

As per the Gauss Seidel method, we write $A = U + D + L$, where U and L are respectively the strict upper and lower triangular matrices of A and D is its diagonal matrix. The Gauss Seidel method is an iterative process, given by:

$$\mathbf{x}_{(k+1)} = (L + D)^{-1}(\mathbf{b} - U\mathbf{x}_{(k)}),$$

for a linear equation $A\mathbf{x} = \mathbf{b}$. Here, $\mathbf{x}_k$ will converge to the solution of this linear system under certain conditions. Also, note the form of this iteration allows us to proceed component-wise, using values of $\mathbf{x}_{(k)}$ that have already been calculated in the current iteration.

The Hansen-Sengupta operator is then a single step of the Gauss Seidel iteration:

$$\mathbf{Y} = \mathbf{x} - D^{-1}\left\{\mathbf{l} + L(\mathbf{X}' - \mathbf{x_c}) + U(\mathbf{X} - \mathbf{x_c})\right\}, \qquad \mathbf{X}' = \mathbf{Y} \cap \mathbf{X}. \tag{2.25}$$

In (2.25), we emphasise the component-wise nature of this iteration. Whenever we calculate a component Y_i of $\mathbf{Y}$ we immediately update the value of $\mathbf{X}'$ appropriately before we compute the remaining components of $\mathbf{Y}$. A difficulty arises when one of the diagonal elements of D contain a zero, in which case D^{-1} is not defined. In this situation, Hansen and Sengupta recommend the use of extended interval arithmetic to compute the result.

The use of this operator (which we denote by $H(\mathbf{X})$) as an inclusion test is justified along the same lines as that of the Krawczyk and Newton operators. This is due to Moore and Qi [30].

THEOREM 9. *Let $H(\mathbf{X})$ be defined as above. Then:*
(1) $H(\mathbf{X}) \subseteq \mathbf{X}$ implies the existence of a zero of $\mathbf{f}$ in $H(\mathbf{X})$.
(2) If $H(\mathbf{X}) \subset \mathbf{X}$ then this zero is unique.
(3) If $H(\mathbf{X}) \cap \mathbf{X} = \emptyset$, then $\mathbf{X}$ has no zero of $\mathbf{f}$.

Proof. The proof is along the same lines as the proofs for the Krawczyk / Interval Newton operators and can be found in [34] or [30]. As before, the proof centres around the existence of a fixed point of a mapping defined in such a way that all its fixed points are roots of $\mathbf{f}$. Further, [34] shows that this operator produces interval enclosures for the zeroes of $\mathbf{f}$ that are smaller than (or as wide as) the enclosures produced by $K(\mathbf{X})$ from the previous section. **Q.E.D.**

2.2.9 Summary

We have now defined and given the theoretical basis behind an interval arithmetic based exclusion predicate, and three related interval arithmetic based inclusion predicates. Recall that all three of these predicates have the same form; for a box B if $P(B) \subseteq B$ (or $P(B) \subset B$) then B is isolating. Here $P(B)$ is one of $K(B), N(B), H(B)$ defined by equations (2.21), (2.16) and (2.25) respectively.

In addition, for all three of these operators we showed that if a box B contains a root, then so does $P(B)$. This allows us to replace B with $B \cap P(B)$ before subdividing it. Further, if $B \cap P(B) = \emptyset$ then B can be discarded. We now have a completely specified subdivision based method that we can use to isolate the roots of a complex polynomial using interval arithmetic.

2.3 Complex analysis based predicates

In the previous section, we discussed a set of interval arithmetic based predicates that could be used in our subdivision algorithm. Recall that these predicates transformed our problem into that of solving a system of two non linear equations in two variables. We note that this transformation of the problem makes it more general. This is because complex polynomials being holomorphic[3] are continuous and differentiable in the complex plane, and this is a stronger criterion than u and v being continuous and differentiable by themselves. For instance, the derivatives of u and v are related via the Cauchy-Riemann equations:

$$u_x = v_y, \quad u_y = -v_x.$$

Predicates designed using complex analysis can therefore take advantage of properties of f that cannot be exploited by the methods of the previous section. On the other hand, those methods are more general, and can solve **any** system of non linear equations, as long as each of the constituent functions are continuous and differentiable.

We discuss in this section an exclusion predicate, which is due to Yakoubsohn and Dedieu [10, 42] and an inclusion predicate which can be viewed as a sort of a "sign test" which is due to Sagraloff and Yap [37].

2.3.1 The exclusion predicate

We start with the exclusion predicate. The predicate is based on the following function $M(x,t)$ as per Yakoubsohn [42]:

$$M(x,t) := |f(x)| - \sum_{k\geq 1} \frac{|f^{(k)}(x)|}{k!} t^k, \tag{2.26}$$

where $x \in \mathbb{C}$ and $t \in \mathbb{R}^+$. This function is linked closely with the complex Taylor series of the function f about the point x which is given by:

$$f(z) = f(x) + \frac{f'(x)(z-x)}{1!} + \frac{f''(x)(z-x)^2}{2!} \cdots \frac{f^{(k)}(x)(z-x)^k}{k!} \cdots.$$

From the triangle inequality, we can show that:

$$|f(z)| \geq |f(x)| - \sum_{k\geq 1} \frac{|f^{(k)}(x)|}{k!} |z-x|^k = M(x, |z-x|). \tag{2.27}$$

Also, from the definition of $M(x,t)$ (2.26), M decreases as the value of t increases, as $t \in \mathbb{R}^+$. We are now ready to show how this function can be used as an exclusion predicate.

For a box B, consider the function $M(m(B),t)$ defined for a function f. It is related to the Taylor series expansion of f about the centre of the box B. Keeping (2.27) in mind, we consider the four corners of the box B. These are the points whose distance from the centre of the box B is maximum, and this distance is equal to $r(B)$. Since $M(x,t)$ decreases with increasing t, for any $z \in B$:

$$|f(z)| \geq M(m(B), |z - m(B)|) \geq M(m(B), r(B))$$

It follows that if $M(m(B), r(B)) > 0$, then $|f(z)| > 0$ over B and f cannot have a root in B. Sagraloff and Yap in [37] use a slightly more generalised version of (2.26) to deduce additional properties of B. They define, for an $m \in \mathbb{C}$ and $K, r \in \mathbb{R}^+$ the predicate T_K^f:

$$T_K^f(m,r): \quad |f(m)| > K \sum_{k\geq 1} \left| \frac{f^{(k)}(m)}{k!} \right| r^k. \tag{2.28}$$

[3]A holomorphic function is one that is (complex) differentiable in a neighbourhood of every element of its domain

If f is replaced by f', then the test (2.28) is turned into:

$$T_K^{f'}(m,r): \quad |f'(m)| > K \sum_{k\geq 2} \left| \frac{f^{(k)}(m)}{(k-1)!} \right| r^{k-1}. \tag{2.29}$$

Their notation reinforces the fact that T is associated with f. In their usage, m, r are usually related to the midpoint and radius $m(B)$, $r(B)$ of some box B. Further, since we deal here with a single function f whose roots we wish to isolate, they suggest that f can be made implicit in the notation. Then, T_K^f can simply be written as T_K and $T_K^{f'}$ as T'_K. We will use this notation for the rest of this section. In this notation, our exclusion predicate becomes:

```
1.   If $T_1(m(B), r(B))$ holds:
2.       Discard $B$
```

2.3.2 A simple inclusion predicate

Yakoubsohn and Dedieu in their work [42, 10] deal with that they call a "Bisection Exclusion algorithm". This can be viewed as a generalised subdivision algorithm that uses only an exclusion predicate, and no inclusion predicate. They use the exclusion predicate $T_1(m,r)$ as defined by (2.28) and a very simple "inclusion" predicate to include boxes:

$$w(B) \leq \varepsilon, \quad \varepsilon \in \mathbb{R}^+.$$

Note that this is a slight abuse of terminology. Unlike the other inclusion predicates that we studied, this one does not guarantee that a box will be isolating.

However, its use with an exclusion predicate does guarantee that all roots of f (if any) will lie within the union of the output boxes. In other words, most parts of the starting region B_0 that do not contain roots of f will be discarded.

This method can therefore be viewed as an "approximate" method, since it makes no guarantees about roots. However, it has considerable utility in that it is straightforward and easy to implement. Moreover, it will work on any analytic function, even one that has roots of multiplicity $m > 1$. Since no inclusion test is performed, except for a simple width comparison, it is efficient as well. This can therefore be used as the first stage in an algorithm that can run elaborate and expensive tests on the output of this stage, which will in most cases be considerably smaller than the starting region.

2.3.3 The 8-Point Test

Sagraloff and Yap in [37] introduce an inclusion predicate that can detect isolating regions for complex polynomials with **no repeated roots**. Besides being cheap to evaluate, their predicate is also easy to implement exactly. They call the combination of the 8-Point test with the exclusion predicate of the previous section the **CEVAL** algorithm. We use this name throughout this work as well.

We provide in this section a brief exposition of their work, though omitting details of their proof beyond those that are necessary to understand the motivation behind this approach. We start with some basics and a definition.

¶5. Basics: We introduce some basic definitions and ideas behind differentiation in the complex plane. A complex valued function $f(z) = u(x,y) + \mathbf{i}v(x,y)$ is said to be differentiable in an open subset S of $\mathbb{C}$ at $z \in S$ if

$$\lim_{h \to 0} \frac{f(z+h) - f(z)}{h} \tag{2.30}$$

exists. When the limit does exist, $f'(z)$ is the value of the limit. In the evaluation of the limit (2.30) note that $z+h$ can tend towards z from any direction in the open disk $D(z,r)$ for some $r > 0$ such that $D(z,r) \subset S$ [4]. This is the analogue of "left hand" and "right hand" limits in real analysis. An important consequence of this property are the Cauchy-Riemann equations. Though the equations are considered a basic result in complex analysis, their exposition here will lend clarity to some of the other results in this section.

Using (2.30) and allowing h to tend towards zero along the x and y axis respectively (purely real and purely imaginary), we have:

$$\begin{aligned} f'(z) &= \lim_{h \to 0} \frac{u(x+h,y) - u(x,y)}{h} + \mathbf{i}\frac{v(x+h,y) - v(x,y)}{h} \\ &= u_x + \mathbf{i}v_x \qquad (h \in \mathbb{R}) \\ f'(z) &= \lim_{h \to 0} \frac{u(x,y+t) - u(x,y)}{\mathbf{i}t} + \mathbf{i}\frac{v(x,y+t) - v(x,y)}{\mathbf{i}t} \\ &= \frac{1}{\mathbf{i}}u_y + v_y = v_y - \mathbf{i}u_y \qquad (h = \mathbf{i}t, t \in \mathbb{R}). \end{aligned}$$

These give us two equivalent forms for $f'(z)$ which are:

$$f'(z) = u_x + \mathbf{i}v_x = (1/\mathbf{i})u_y + v_y. \tag{2.31}$$

Equating the real and imaginary parts of these forms gives us the Cauchy-Riemann equations.

In some sections, we treat u, v as functions of two variables x, y and define the gradient in the natural way, $\nabla u = (u_x, u_y)$ and $\nabla v = (v_x, v_y)$. Since both u, v are continuous and differentiable real valued functions in two variables, the Mean Value Theorem applies. To define it, we treat a complex number $z \in \mathbb{C}$ as a vector in two dimensional space $z = (\mathtt{Re}(z), \mathtt{Im}(z))$. We then proceed in a manner similar to Lemma 3 to state that there must exist a $c \in [a,b]$ such that $u(b) - u(a) = \nabla u(c) \cdot (b-a)$ where $a, b, c \in \mathbb{C}$ are treated as vectors, the gradient ∇u is defined as above and "$\cdot$" denotes a scalar product of two vectors.

¶6. Disk: A disk in the complex plane is defined in the usual way. A disk $D(m,r)$ with $m \in \mathbb{C}$, $r \in \mathbb{R}^+$ is the set of all complex numbers z such that $|m - z| \leq r$. The disk is said to be centred at m and has a radius of r. A disk associated with a box B is the disk with radius $r(B)$ centred at $m(B)$. Obviously in this case $B \subset D(m(B), r(B))$. Also note that the test T_K^f defined in (2.28) can operate on a disk in the obvious way, using the centre and radius of the disk. We refer to this application as $T_K(D)$ (and the corresponding $T'_K(D)$).

It is also useful to talk about some definitions relating to the boundary of a disk D. The **main compass points** of $D(m,r)$ are the 8 points $m + r \cdot e^{\mathbf{i}j\pi/4}$ for $j = 0, 1, \ldots, 7$. They are given standard labels, and comprised of the **cardinal points**: N, S, E, W and the **ordinal points**: NE, SE, SW, NW. The boundary of the disk D is divided into 8 arcs by these compass points, each arc being described by:

$$A_j := \left\{ m + re^{\mathbf{i}\theta} : j\pi/4 \leq \theta < (j+1)\pi/4 \right\} \quad j = \{0, 1, \ldots, 7\}.$$

For instance, the arc A_0 has E, NE as its end points and the arc A_1 has NE, N and so on.

We now show some results regarding the existence and the number of roots in a disk $D(m,r)$.

[4] For any open set in the complex plane, such a disk is guaranteed to exist.

Lemma 10. *For any disk D, if $T_1(D)$ from (2.28) holds, then D contains no zeroes of f.*

Proof. The proof is exactly the same in the case of a box. In the case of a disk, the points that are at a maximum distance from the centre lie on the perimeter of the disk. **Q.E.D.**

The second lemma is similar, but gives an upper bound on the number of roots a disk can contain, and is due to Sagraloff and Yap (see Lemma 6,7,8 of [37]):

Lemma 11. *For any disk D, if $T'_{\sqrt{2}}(D)$ from (2.29) holds, then D contains at most one root of f.*

Proof. We will first illustrate how $T'_{\sqrt{2}}(D)$ imposes restrictions on the range and behaviour of $f'(z)$ across a disk $D(m,r)$. Let $L = \sum\limits_{k\geq 2} \left| \frac{f^{(k)}(m)}{(k-1)!} r^{k-1} \right|$. For some $z_1 \in D(m,r)$, we can use the Taylor series expansion to write

$$f'(z_1) = f'(m) + \sum_{k\geq 2} \frac{f^{(k)}(m)}{(k-1)!}(z_1 - m)^{k-1},$$

and from the triangle inequality, $|f'(z_1) - f'(m)| \leq L$. The maximum value of $|f'(z_1) - f'(m)|$ is exactly L, and the points at which this value is attained will lie on a circle of radius L centred at m. Considering the triangle with vertices at $f'(m)$, $f'(z_1)$ and the origin we have:

$$|\arg f'(z_1) - \arg f'(m)| \leq \arcsin\left(\frac{L}{f'(m)}\right).$$

This follows from the fact that when the LHS reaches its maximum value, $f'(z_1)$ will lie on the circumference of the disk, and the triangle will be right angled at z_1. From our predicate $T'_{\sqrt{2}}$ we have $|f'(m)| > \sqrt{2}L$ and as a consequence, we must have $|\arg f'(z_1) - \arg f'(m)| < \arcsin(\frac{1}{\sqrt{2}}) = \frac{\pi}{4}$. (Note that the ">" from T' makes the inequality strict). This tells us that for any $z_1 \in D(m,r)$, the argument of $f'(z_1)$ can differ by at most $\frac{\pi}{4}$ from that of the centre m. What this also implies is that for any two $z_1, z_2 \in D(m,r)$ we must have

$$|\arg f'(z_1) - \arg f'(z_2)| < 2.\frac{\pi}{4} = \frac{\pi}{2}. \tag{2.32}$$

From (2.31) we have $\arg f'(z) = \arctan \frac{v_x}{u_x} = -\arctan \frac{u_y}{u_x} = -\arg \nabla u(z)$. Applying this to (2.32) we have:

$$|\arg \nabla u(z_2) - \arg \nabla u(z_1)| < \frac{\pi}{2}. \tag{2.33}$$

We can now prove the main result by contradiction. Let $z_1, z_2 \in D(m,r)$ be two zeroes in the disk $D(m,r)$. Now if $z_1 = z_2$, then $f'(z_1)$ must necessarily be zero. However, $T'_{\sqrt{2}}$ holds, and therefore T'_1 must hold and f' cannot have a zero in $D(m,r)$ so we can assume that $z_1 \neq z_2$.

Now $f(z_1) = f(z_2) = 0$ implies that $u(z_1) = u(z_2) = v(z_1) = v(z_2) = 0$. From the Mean Value Theorem, we must have some $a, b \in [z_1, z_2]$ such that $\nabla u(a) \cdot (z_1 - z_2) = u(z_1) - u(z_2) = 0$ and $\nabla v(b) \cdot (z_1 - z_2) = v(z_1) - v(z_2) = 0$. We conclude from the above that both $\nabla v(b)$ and $\nabla u(a)$ are perpendicular to $(z_1 - z_2)$.

However, $\nabla v(b) = (v_x(b), v_y(b)) = (-u_y(b), u_x(b))$. Also:

$$\nabla v(b) \cdot \nabla u(b) = -u_x(b) \cdot u_y(b) + u_x(b) \cdot u_y(b) = 0.$$

We then conclude that $\nabla u(b)$ is perpendicular to $\nabla v(b)$ and therefore it must be parallel to $(z_1 - z_2)$. This also makes it perpendicular to $\nabla u(a)$. This is a contradiction because the arguments of $\nabla u(z)$ evaluated at two points $z_1, z_2 \in D(m,r)$ must necessarily be less than $\frac{\pi}{2}$ from (2.33). **Q.E.D.**

Both of these lemmas provide us more information about the number of roots in D but neither of them can guarantee that D is isolating. For this purpose, Sagraloff and Yap introduce the 8-point-test which is very roughly a sign change test. The test evaluates f at a fixed number of points on the boundary of D and requires that the real and imaginary parts u, v of $f(z) = u(x,y) + \mathbf{i}v(x,y)$ satisfy certain conditions in order for the disk D to be isolating.

The main concept behind the test is that of an **arc crossing**: u is said to have an arc crossing at an arc A_j if its signs at the end points of A_j differ. (Or if $u = 0$ at one of the end points). More specifically, let $j \in \{0, \ldots, 7\}$. Then there occurs an **arc-wise u-crossing** of $D(m,r)$ **at** A_j if

$$\begin{cases} u(m + re^{\mathbf{i}j\pi/4}) \cdot u(m + re^{\mathbf{i}(j+1)\pi/4}) < 0, \text{ or} \\ u(m + re^{\mathbf{i}j\pi/4}) = 0. \end{cases} \tag{2.34}$$

The 8-point test applied to a disk $D(m,r)$ is said to succeed **if**:

- There are exactly **two** arc-wise u crossings A_j and A_k and exactly **two** arc-wise v crossings $A_{j'}$ and $A_{k'}$ on the boundary of D.
- The crossings **interleave**. We must have either $j < j' < k < k'$ or $j' < j < k' < k$.

We can now state the inclusion test in its entirety. This is essentially Theorem 2 from [37] and is stated here without a complete proof:

THEOREM 12. *Suppose $T'_6(m,4r)$ holds.*
(i) If $D(m,r)$ is isolating, then $D(m,4r)$ passes the 8-point test.
(ii) If $D(m,4r)$ passes the 8-point test, then $D(m,4r)$ is an isolating disk.

Proof outline. In Lemma 10 and Lemma 11 we showed how $T_K(D)$ and $T'_K(D)$ impose conditions on how $|f(z)|$ and $\arg f(z)$ can vary across a disk D. It turns out that they in fact impose stronger conditions than the ones we proved above. T_K constrains the arc crossings of u, v to certain specific areas of the boundary of $D(m,r)$ and imposes an upper bound on the difference between the arguments of those points. The proof structure for these properties is largely geometric, and an exposition of the entire proof is beyond the scope of this thesis. A full proof can be found in [37].

Note that if $D(m,r)$ is isolating then $D(m,4r)$ is guaranteed to pass the 8-point test. However, if $D(m,4r)$ passes the 8-point test, then we can only make the (slightly weaker) statement that the larger disk $D(m,4r)$ is isolating.

With our inclusion predicate in place, we now have a complete subdivision algorithm to isolate the roots of a complex polynomial with no repeated roots. Note that this inclusion test is less general than the exclusion test. While the exclusion test works for any analytic function f, the inclusion test works only polynomials with no multiple roots.

It is clear that in this case, we do not suffer from the issue of roots shared between boundaries of boxes as in Section 2.2.6. Instead, we deal with the notion of a root present in a disk $D(m,4r)$ centred at the centre of a box $B(m,r)$ that we assert is isolating. We might now ask what would happen if two adjacent boxes passed the 8-point test. Clearly, their isolating disks will intersect.

We can deal with this by imposing stronger conditions on the 8-point test as per [37]. Refer to Lemma 11, in which we showed that if $T'_{\sqrt{2}}(m,r)$ holds, then $D(m,r)$ has at most one root. Now, if $T'_{\sqrt{2}}(m,8r)$ held for some box centred at m with radius r, then it would guarantee that even if adjacent boxes were to pass the 8-point test, we can keep *only* the smallest of the disks arising from these boxes. This is because $T'_{\sqrt{2}}(m, 8r)$ guarantees at most one root within the disk $D(m, 8r)$, and all $D(m_i, 4r_i)$ resulting from the adjacent boxes will be contained within it.

With this we are ready to state the algorithm.

```
    INPUT: A box $B$ with centre $m$ and radius $r$.
1.  If $T_1(m, r)$ holds, discard $B$.
2.  Else if $T'_6(m, 4r)$ and $T'_{\sqrt{2}}(m, 8r)$ hold:
3.      If $D(m, 4r)$ fails the 8-point test, discard $B$.
4.      Else
5.          If $D(m, 4r)$ intersects with an existing output disk $D'$
6.              Replace $D'$ with the smaller of $D'$ and $D$.
7.          Else insert $D(m, 4r)$ into the output.
8.  Else
9.      Split $B$ into four equal children and insert them into $Q$.
```

The final point to note is that the output consists of a list of disks $D(m, r)$ and not boxes. Also for a box B, we can only state that $D(m, 4r)$ is isolating, and that is a larger disk than the box. This implies that the output is a set of disks whose **centres** are within the initial region B_0, and the roots they isolate need not necessarily lie within B_0.

2.4 Other approaches

2.4.1 Introduction

This section discusses various other approaches to root finding that do not fit into the general structure of subdivision based algorithms. We limit our discussion to methods that allow us to isolate **all** roots of a given polynomial (including its complex roots) and not just those that are real. For the problem of finding *only* real roots, a specialised set of methods are applicable including those based on Descartes' rule of sign or on Sturm sequences (See for example Chapters 6 and 7 of [43] or the excellent summary of history and recent progress in this area by Pan [45]). Our discussion is further limited to approaches that are known to be stable and robust, and are implemented in popular mathematical packages.

Note that the approaches below are discussed for the sake of completeness, and our main motivation is to compare various **subdivision** based methods of root isolation. Our implementation efforts and results along those lines are discussed in subsequent chapters and the reader can skip this section without any loss of continuity.

2.4.2 Weierstrass type iterations

This section discusses two approaches: The Weierstrass Durand-Kerner iteration [23] and the closely related Aberth-Erlich iteration [1]. Both of these are simultaneous iterations meaning that at each step of the iteration, successively better approximations of **all** roots of a polynomial p are calculated. Both approaches start by computing initial estimates of the roots of p. The problem of computing the initial root estimates is an area of research by itself, but even simplistic estimates suffice. One such set of estimates for a polynomial of degree n are the n complex roots of unity [4]. From the initial estimates, these algorithms calculate successively better approximations of all roots until a termination condition is reached.

Consider the polynomial p given by

$$p(z) = \sum_{i=1}^{n} a_i z^i = a_n \prod_{i=1}^{n} (z - \alpha_i),$$

where each α_i is a root of $p(z)$. We evaluate the derivative of p based on its factorised form:

$$p'(z) = a_n \sum_{i=1}^{n} \left(\prod_{j \neq i} (z - \alpha_j) \right).$$

Now, if α_i is a root of p, $p'(\alpha_i)$ will be given by

$$p'(\alpha_i) = a_n \prod_{j=1, j \neq i}^{n} (\alpha_i - \alpha_j).$$

This can be "plugged in"[5] to the Newton iteration to give the Weierstrass Durand-Kerner iteration, which is:

$$x_i^{k+1} = x_i^k - \frac{p(x_i^k)}{a_n \prod\limits_{j=1, j \neq i}^{n} (x_i^k - x_j^k)},$$

where x_i^k in general is the i'th root of $p(z)$ on the k'th iteration step. Replacing $p(z)$ with

$$r_i(z) := \frac{p(z)}{\prod\limits_{j=1, j \neq i}^{n} (z - \alpha_j)},$$

into the Newton iteration gives us the Aberth-Erlich method. Note that $r_i(z)$ is the linear factor $(z - \alpha_i)$ of p, corresponding to the root α_i. The zero of this linear factor $(z - \alpha_i)$ is one of the roots of p. When we have only estimates of the roots, then the r_i terms are correspondingly estimates of the factors associated with each root. The Aberth-Erlich iteration is then given by:

$$x_i^{k+1} = x_i^k - \frac{(p(x_i^k)/p'(x_i^k))}{1 - (p(x_i^k)/p'(x_i^k)) \sum\limits_{j=1}^{n} 1/(x_i^k - x_j^k)}.$$

This iteration is the focus of Bini's and Fiorentino's work on MPSolve [4, 6]. The MPSolve package is referred to in latter sections of this thesis as it is used to compare the relative speed of subdivision based approaches to those based on simultaneous iteration.

2.4.3 The Jenkins-Traub algorithm

Jenkins and Traub in [21] introduced a globally convergent algorithm to calculate all roots of a polynomial p using a three stage process. The algorithm is known to be extremely stable and robust, and is in use in various commercial packages [45]. The algorithm does not attempt to find all roots of p simultaneously. Instead, it approximates one of the n roots α_i of p and then proceeds to find all $(n-1)$ roots of $p/(z - \alpha_i)$ and so on. Roots are found in generally increasing order of magnitude [21]. The basic idea behind the algorithm is to construct a sequence of polynomials H_λ which are meant to converge to the polynomial $p/(z - \alpha_i)$ which is the cofactor of the root α_i. The iteration step of the Jenkins-Traub algorithm (which [21] calls "Stage 3: The variable shift process") is very closely related to the Newton iteration of the previous section, and is given by:

$$s_{\lambda+1} = s_\lambda - \frac{p(s_\lambda)}{H_{(\lambda+1)}(s_\lambda)}.$$

[5] In reality, its not as simple as just plugging it in. Proving that it is in fact convergent and that it actually works is a far more complex and tedious task.

Here, each s_λ is the current estimate of a root, and $H_{(\lambda+1)}$ is the current estimate of the cofactor of that root. This can be viewed as an iteration on a rational function $p(z)/C(z)$ where $C(z)$ can be made as close to the cofactor of the desired root as possible.

One of the disadvantages of the approach it is difficult to apply to the problem of isolating roots within a region of interest, a task to which subdivision algorithms are particularly well suited. Further, the roots need to be found in roughly increasing order of magnitude to avoid numerical stability issues when deflating p with a large root [21]. Further, Goedecker in [13] points out that this method is generally unsuitable for use for polynomials of degree $n > 50$ and speaks of convergence issues around multiple roots and clusters of roots. Nevertheless, its stability and speed of convergence make it one of the prime choices for numerical packages.

Chapter 3

Implementation

3.1 Introduction

The previous chapter discussed root isolation from a purely mathematical viewpoint. Implementing an algorithm that has its basis in analysis poses its own challenges and is an area of study in its own right. The most basic concept in real analysis, the set of real numbers $\mathbb{R}$ is incapable of being represented by a digital computer with discrete and finite memory. In most cases, computers can only work with approximations of its members. Implementers of numerical algorithms also need to deal with issues around **precision**, **overflow** and employing the correct form of **rounding**. This chapter discusses some of these issues in the context of subdivision algorithms and also serves to provide an outline of our implementation.

We choose to implement our algorithms in `C++` for multiple reasons. It is the most commonly used language for scientific computing and has a large and mature body of open source code that can be built upon. We describe some of the open source libraries that we used through the rest of this section. Further, most existing root finders are written in `C`, and using `C++` will allow a fair comparison of our work with other approaches.

3.2 Machine arithmetic

In our mathematical definitions, we dealt with various sets of numbers. The most common being the real numbers $\mathbb{R}$ and the complex numbers $\mathbb{C}$. Both of these sets are dense, infinite and uncountable. Computers with finite memory cannot represent every member of either of these sets, and in most cases can store only approximations. The set of integers $\mathbb{Z}$ is countable but still infinite, and though every member of this set is capable of an exact representation, a computer with finite memory can represent only a finite subset of the integers. We deal with the machine representation of each of these sets below.

3.2.1 Fixed precision formats

These representations use a **fixed number** of bits to represent a given number. The number of bits used in this representation is often referred to as the *precision* of the type[1]. This is usually chosen to be the register size of the underlying hardware, in which case the format is said to be at “machine precision”. In this case, standard arithmetic operations can be performed using dedicated hardware and as a consequence are very fast. This is in contrast to **extended precision** formats which are discussed in Section 3.2.2.

[1]This definition tends to vary, some definitions include the sign and exponent bits as well, others choosing just the significand.

¶7. Integral types: `C++` provides two **fixed precision** integer types, the `int` and the `long`. The length of an `int` (resp. `long`) is implementation dependent and all the `C++` standard (Page 53. [20]) requires is that it can store all values in the range `[INT_MIN, INT_MAX]` (resp. `[LONG_MIN, LONG_MAX]`) as defined in the standard header `<climits.h>`. On most 32 bit architectures, the `int` is 32 bits (4 bytes) in length as is a `long`[2].

Basic arithmetic operations on these fixed precision integer types are often performed in hardware by dedicated circuitry and are extremely fast. The operators $\{+, -, \cdot\}$ on integral types are always exact as long as the results do not overflow. The $\div$ operator on integers is not closed and in `C++` will return the integer quotient of the division.

¶8. Floating point types: `C++` provides a single precision floating point type `float` and a double precision floating point type `double` that occupy 32 and 64 bits respectively. Their behaviour is governed by the IEEE-754-1985 standard (and the IEEE-754-2008 standard that succeeded it [19]). In most of our work, we use the double precision type and we now devote a short paragraph to the details behind it.

A 64 bit double precision floating point type (the standard calls this binary64, Section 3.1 [19]) is made up of three components, a single sign bit, a 11 bit exponent and a 52 bit significand. The exponent is encoded in bias offset form, meaning that a fixed *exponent bias* value is subtracted from it (Section 3.5.2 [19]) to obtain the actual exponent. The number is then given by

$$n = (-1)^{sign} \cdot 2^{exponent-bias} \cdot 1.m_1m_2m_3 \dots m_n,$$

where $m_1m_2m_3 \dots m_n$ are the significand bits. The significand is stored in normalised form, which means its bits are shifted left as far as possible such that the first digit is always 1. The first digit is implicit $(1.m_1 \dots)$ and raises the number of significand bits to 53. (This is not always the case, see the section on underflow, below).

¶9. Overflow: When the result of a floating point or integral calculation is such that its minimal representation requires a precision larger than the precision level of the calculation, an overflow is said to have occurred. In the case of the integral types, this happens precisely when the result does not fall in the range `[INT_MIN, INT_MAX]` (or `[LONG_MIN, LONG_MAX]`). For floating point types, an overflow occurs when the exponent is larger than the largest exponent that can fit in 11 bits. An overflow is a serious problem, and stable numerical method implementations should be able to detect it and take mitigating steps or signal that the method must halt.

¶10. Underflow: Consider the case of a floating point calculation whose result is positive. When the result is sufficiently close to zero on the positive axis, its exponent might be smaller than the smallest exponent that can be represented by the type. When this happens, an underflow is said to have occurred[3]. This leads to the so called *zero gap*, the gap between the closest real number capable of a fixed length floating point representation and 0 is larger than the gap between any two other adjacent numbers that can be represented by the type.

The designers of the IEEE-754 standard mitigated this issue by introducing the subnormal numbers. These are numbers whose implicit bit is 0 instead of the usual 1. This allows sig-

[2]The fact that a `long` must be "at least as long as" ([20, Page 54]) an `int` along with the fact that `[LONG_MIN, LONG_MAX]` can be represented in 32 bits in twos complement form means that on most 32 bit architectures, a long is the same size as int. This is an endless source of confusion for those new to `C++`.

[3]Note that this is different from being smaller than the smaller representable number of the type as all negative numbers are smaller

nificands of the form $0.m_1m_2m_3m_4..m_n$, that gradually lose precision as they get smaller (the exponent will remain fixed at its smallest possible value)[4]

¶11. Rounding: The floating point types described above are fixed precision, and as a result are capable of representing only a very small subset of the real numbers. Numbers that cannot be represented must be rounded to the number of bits of precision available. Results of calculations are represented as if the calculation occurred at infinite precision and the results were rounded to the significand precision. The IEEE-754 standard specifies a set of rounding types for all floating point operations (Section 4.3 [19]):

- Round to nearest: Rounds the number to the nearest representable value. If the number falls midway between to representable values (a tie), we can chose either the value with a zero LSB[5] (the default rounding algorithm) or one with a LSB of 1. Note that ties are broken consistently one way or the other, and not by a random choice.
- Round towards 0: Rounds to the nearest representable number between the result and zero.
- Round towards $+\infty$: Rounds to the nearest representable number larger than the result.
- Round towards $-\infty$: Rounds to the nearest representable number smaller than the result.

Each of these rounding modes have their own utility. For instance, while evaluating expressions involving intervals, we would round operations involving the lower limit of the interval towards $-\infty$ and those involving the upper limit towards $+\infty$ in order to produce a wider interval that is a superset of the (infinite precision) result. Note that this fits in well with our interpretation of an interval as the upper and lower bound on the result of a calculation.

3.2.2 Extended precision formats

Though the precisions and sizes described in the previous section suffice for a large category of mathematical programs, some require higher precision than the above representations are capable of. We therefore need formats that have a higher number of bits than the machine precision formats of the previous section. The IEEE-754 standard (Section 3.7 [19]) calls these *extended precision formats*. These are also called arbitrary precision formats, though this term is misleading because the precision is always bounded by the available system memory and implementation. The programming community tends to prefer the less formal name "bignums" for such formats.

The `C++` standard library does not provide an extended precision implementation and this space is filled by multiple open source libraries. The most widely used of these libraries is the GNU Multi-Precision Library [11] (**GMP**). It is a mature, well maintained library used in a wide variety of open source projects and is backed by an active developer community. This makes it the obvious choice for our work. We provide a brief introduction to some of the representations GMP offers that are central to our implementation.

¶12. Integers: The GMP extended precision integer type `mpz_t` is represented as a sign and a magnitude [12]. An integer is stored as an array of "limbs", each limb being the size of a machine word (32 or 64 bits). The limbs are dynamically allocated according to the magnitude of the integer, and the number of limbs is limited only by the available system memory.

[4]For an interesting discussion of these issues, see [22]

[5]LSB is an acronym for the Least Significant Bit

¶13. Floating point numbers: The GMP floating point type `mpf_t` like the integer types is stored as an array of limbs. In addition, an exponent and a target precision (in terms of limbs) are associated with every floating point representation. The exponent is a machine precision `long`, and is not stored in the bias offset format. Results are computed until the number of limbs specified by the precision. The precision of a floating point type is limited only by the available system memory, and the exponent is limited to the range `[LONG_MIN, LONG_MAX]`.

¶14. Rationals: A rational type `mpq_t` is stored as an `mpz_t` pair representing a numerator and a denominator. The canonical form always contains a positive denominator, and the numerator and denominator always contain no common factors.

¶15. Rounding: The GMP library does not offer any guarantees that results involving floating point types will be rounded consistently to the desired precision, instead stating that the results will be correctly rounded in "most cases" and truncated in others [12]. Applications that desire correct and consistent rounding must use libraries like the GNU MPFR [31]. MPFR stays true to the spirit of the IEEE-754 standard in that it supports all of the rounding modes that the standard outlines with the same semantics. Further, when the target precision is 53 bits (the same as the machine `double`), MPFR will accurately reproduce every operation as if it was performed with corresponding machine type [32].

3.2.3 The dyadic numbers

It turns out that the set of numbers that can be represented exactly using a finite binary floating point representation (`mpf_t`) is very small, and the elements of of this set are called the **dyadic numbers**. They are of the form $a.2^{-e}$ where $a, e \in \mathbb{Z}$. For any number $r \in \mathbb{R}$ that is not dyadic, we can construct the sequence

$$\frac{\lfloor 2^k.r \rfloor}{2^k} \qquad k = 0, 1 \ldots$$

whose elements are successively better approximations of r as k increases. Hence, any number in $\mathbb{R}$ can be represented by a dyadic number that is an arbitrarily small distance away from it. However, as this distance decreases the precision and the memory required to store the approximation increase.

It can be seen that the set is closed under the operations $\{+, -, \cdot\}$. For example,

$$\frac{p}{2^a} + \frac{q}{2^b} = \frac{p.2^{b-a} + q}{2^b}, \quad b \geq a$$

and

$$\frac{p}{2^a} \cdot \frac{q}{2^b} = \frac{p \cdot q}{2^{(a+b)}}.$$

Therefore, the results of these operations on dyadic numbers can be represented exactly, and will in fact be exact if the target precision is high enough. The dyadic numbers are not closed under the operator $\div$ and the quotient of two dyadic numbers is not necessarily dyadic. Hence, the result of the $\div$ operator may not have a finite binary representation.

3.3 The Core Library

3.3.1 An introduction to Core

The Core library [9] is a collection of C++ classes for exact computation with algebraic real numbers and arbitrary precision arithmetic[6]. It is built over the GMP and MPFR libraries that we introduced in Section 3.1, and in addition to providing a unified and consistent C++ API over the mpx_t types, provides a useful set of extensions to perform tasks such as polynomial representation and evaluation, basic linear algebra, real root isolation and so on. In addition, Core defines multiple **levels** of operation over which a program can be compiled and executed. Each of these levels provide stronger guarantees on exactness at the cost of slower execution. For a detailed description of the design and implementation of Core, see [46].

Our complex root isolation algorithms have been implemented using the Core library as this allowed us to build upon its existing polynomial evaluation and linear algebra code and to use the clean Core API over the GMP types.

During the course of our implementation, we wrote a framework of classes that are of considerable utility even outside of our work. These include a templated interval arithmetic package, a complex number type and a consolidated API over machine doubles. In addition, we improved parts of the Core library by contributing specific algorithms not directly related to our work, such as an implementation of the Bareiss algorithm (See Chapter 10 in [43]) for determinant calculation and matrix inversion.

All of this work has been reviewed and submitted to the Core repository and forms a part of the latest Core release (Version 2.1, which can be downloaded at [8]).

3.3.2 The Core design

One of the aims of the Core library is to make it easy to write programs that can be run at different levels of precision without significant changes to the algorithm code. For instance, a program that operates using the machine precision types long and double with minor changes should be able to use the GMP types mpz_t and mpf_t if the calculation it performs needs more precision than the machine types can offer. Programs that use the Core library specify a "CORE_LEVEL" which the C++ preprocessor uses to perform certain code substitutions. Take the following code sample for instance:

```
#define CORE_LEVEL 2
#include "Core/Core.h"

int main(int argc, char **argv) {
  double a = 5.0, b = 4.0;
  long d = 56;
  double c = a*b;
}
```

The sample defines a CORE_LEVEL of 2. Based on the Core level that is defined, and on definitions in Core.h, the pre-processor will replace all occurrences of double with the default floating point type corresponding to that level, and similarly for occurrences of long integral types. The program might end up looking like the following.

```
int main(int argc, char **argv) {
  BigFloat a = 5.0, b = 4.0;
  BigInt d = 56;
```

[6]An algebraic number is a number is the root of a non zero univariate polynomial with integer coefficients. It is important note that though an algebraic number may not be dyadic, it can be represented in terms of its "construction" from other dyadic numbers.

```
  BigFloat c = a*b;
}
```

Here, `BigFloat`, `BigInt` are classes defined by Core in a manner that allows them to replace the basic types `double` and `long` in all situations. For instance, the snippet above depends on the following definitions:

```
class BigFloat {
  // Implicit conversion from a machine precision
  // double.
  BigFloat(const double &);
};
// To substitute the "*"
BigFloat operator*(const BigFloat &l,
                   const BigFloat &r);
```

It is important to note that `int` and `float` types are never substituted by default, and if the user wants to prevent an automatic substitution he can explicitly do so by using the typedefs `machine_double` and `machine_long`. We now describe the substitutions that occur at each core Level and the associated precision of the calculations.

Also, note that these substitutions are performed by the `C` preprocessor and therefore take place at compile time. If programs wish to switch their Core Level dynamically at run time, it involves slightly more effort and operation at Level 4, which is explained below.

Level 1: At Level 1, all calculations occur at machine precision by default and no preprocessor substitutions occur. Due to the fact that these operations occur in hardware, this level is much faster than the other Core levels. One of our contributions to Core was the introduction of Wrapper types at Level 1 to allow a greater degree of interoperability with the other core Levels.

Level 2: At Level 2, `double` and `long` are substituted with the Core classes `BigFloat` and `BigInt` respectively. These are wrappers over the GMP types `mpf_t` and `mpz_t` through the MPFR API that ensures correct rounding. Further, Core can automatically determine the required target precision for a calculation such that if the operands are dyadic, then the result is represented exactly for the operators $\{+, -, \cdot\}$. Core also introduces a `BigRat` class as a wrapper over `mpq_t`.

Level 3: Level 3 provides "**Exact Numerical Computation**" as defined by [44] through the `Expr` class using the theory of *constructive zero bounds* (See Chapter 12 in [43]). The details behind this approach are beyond the scope of this thesis, and can be found in [46]. Each `Expr` numeric type stores a directed acyclic graph that describes its construction from other algebraic constants. The `Expr` type supports **exact** comparisons (not just ε correctness) for algebraic numbers, and an associated approximate comparison for the transcendentals.

Level 4: Level 4 allows for mixed mode operation. Machine precision types `long`, `double` can be used alongside `BigFloat`, `BigInt` and `Expr` types. All references to types must be explicit and no preprocessor substitutions occur. This mode is useful for programs that need to switch to a higher precision only in certain situations. Such programs can operate at Level 1 by default, and if they detect that they require more precision, convert their working set into a higher level type and proceed with execution. This strategy allows for faster operation since extended precision types are used only if required, and not by default.

There are multiple programming methods that can be used to achieve this. For instance, we could write templated versions of our algorithms parametrised by the number type they operate on. The following code sample should make this clearer.

```
#define CORE_LEVEL 4
template<typename NT> class Algorithm {...};

void Run() {
  Algorithm<machine_double>::run();
  ..
  if (needsMorePrecision()) {
    ConvertWorkingSet();
    Algorithm<BigFloat>::run();
  }
}
```

We use a similar approach in our implementation of the Newton type methods. Specifically, in our treatment of unresolved boxes (see Section 3.4.4).

3.3.3 Support for Interval arithmetic

One of our contributions to Core (as of version 2.1) is a templated interval arithmetic class:

```
template<typename NT> class IntervalT {
  // Construct from end points.
  Interval(const NT &left, const NT &right);
  // Construct from degenerate
  Interval(const NT &left);
};
```

The class can be parametrised with any of the number types from any of the Core levels. An instance of the `IntervalT<NT>` class can be used in most places where a standard number type is used since it implements the interval equivalents of the basic operators $\{+, -, \cdot, \div\}$. Further, floating point numbers are automatically (type) upgraded to the equivalent degenerate interval.

At **Level 1**, the interval endpoints are machine precision floating point numbers. Like all other operations, results of interval operations at Level 1 might be inexact and for all operations, interval upper and lower bounds are rounded upward and downwards respectively. Note that this means that a result interval might be wider than the corresponding infinite precision result[7].

At **Level 2**, the interval endpoints are instances of Core `BigFloat` class. As a consequence, the operations $\{+, -, \cdot\}$ are exact for dyadic endpoints. The operator $\div$ may not be exact, but our implementation rounds the endpoints of the resulting interval correctly so that the resulting interval contains the exact result. Some authors refer to the formal system in which all interval endpoints are fixed precision floating point types as rounded interval arithmetic (See Section 3.2 of [28]). This approach is not quite the same, since we allow for all operations except for $\div$ to be exact on dyadic intervals.

One last point to note is that at as a side effect of implementing the API of the number types, the Core polynomial package can evaluate a polynomial $p(x)$ for an Interval valued x. The polynomial package defines

```
template <typename T> class Polynomial {
  // Note that this is a template member function. T is the
  // type parameter for the polynomial coefficients, and NT
  // the type parameter for the point at which the polynomial
```

[7]To round intervals at machine precision, we require access to the floating point environment as defined in the `C` standard header `<fenv.h>`. On platforms that do not implement this header, intervals will be rounded to the IEEE default.

```
  // is being evaluated. Note that T and NT must either be
  // interoperable or implicitly convertible.
  template <typename NT> NT eval(const NT &x);
};
```

which the compiler can instantiate for `NT=IntervalT`.

3.3.4 Support for Complex numbers

As part of the groundwork for implementing the complex analysis based predicates (Section 2.3) we implemented a Complex number type which is now a part of Core (as of Version 2.1).

```
// The cartesian form of a complex number.
template <typename NT> class ComplexT;
// The polar form of a complex number.
template <typename NT> class PolarComplexT;
```

Here, `NT` can be any of the types discussed earlier. These classes are used extensively in our implementation of the CEVAL algorithm. Polar and Cartesian complex numbers are interoperable, but the conversion between these two types is inexact since it involves the evaluation of trigonometric functions.

We note that the `C++` standard now defines a Complex number type as part of the standard library (Section 26.2 in [20]). The reason for not using it is that though it is a part of the standard, it is not implemented on all the platforms that Core supports. Furthermore, we provide an API to select specific rounding modes and match the API of the Core real number types.

3.3.5 Working with polynomials

The most basic of our requirements is the ability to represent and operate on polynomials. For this, we use the Core polynomial library which offers classes that represent and implement operations on univariate and bivariate polynomials.

¶16. Polynomial representation: Consider the type `Polynomial<NT>` parametrised with `NT`, the number type of the polynomial coefficients. The most natural machine representation of a univariate polynomial of degree n is a vector of size $n+1$ that stores each of its $n+1$ coefficients (this includes the coefficient of z^0, the constant term).

Bivariate polynomials (the `BiPoly<NT>` class) can be represented in much the same way. Let $p(x, y)$ be a bivariate polynomial of y-degree n. This can be written as:

$$p(x,y) = \sum_{i=0}^{n} y^i f_i(x),$$

where $f_i(x)$ is a univariate polynomial in x. Hence, a bivariate polynomial is represented as a vector of univariate polynomials, each member of the vector being implicitly a coefficient of some power of y.

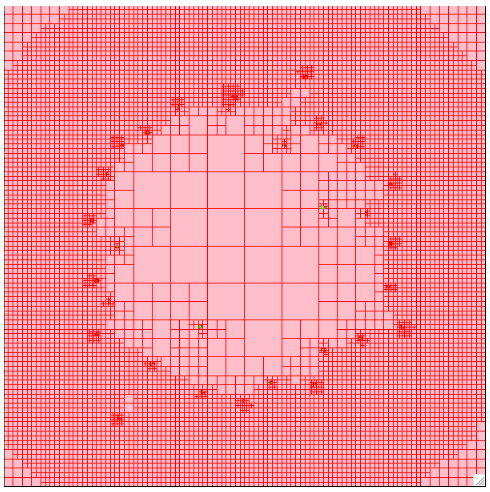
We also make a short note on the input format for polynomials. Polynomials can be fully specified either at the program command line, or as a reference to a file. We use the same file format as the MPSolve package [6], details can be found at [5].

¶17. Separating $p(z)$ into $u(x,y)+\mathbf{i}v(x,y)$: Recall that our Newton type operators work by considering the system of equations $u(x,y) = v(x,y) = 0$ where u, v are the real and imaginary parts of $f(z)$. We therefore need a method to convert a univariate polynomial into two bivariate polynomials that represent its real and imaginary parts.

When the coefficients of f are real, this straightforward to do. We employ a method that calculates the composition of f with the polynomial $t(x, y) = x + y$. This composition is a bivariate polynomial in x, y. Obviously, terms with even powers of y will belong to u and those with odd powers of y will belong to v (with the appropriate sign). The bivariate polynomial representation above makes it even easier to compute this separation.

3.3.6 Visualising subdivision trees

In some cases, we might be interested in inspecting the subdivision tree that our algorithm generates in addition to its actual output. This is helpful in debugging, and also gives us a visual indication of how our predicates are performing. With that in mind, we implemented a display module in `C++` using the `OpenGL` graphics library to visualise this subdivision tree, an example of which is given in Figure 3.1. Even though we do not require 3D graphics, we use


(a) The roots of a degree $n = 25$ polynomial (using CEVAL).

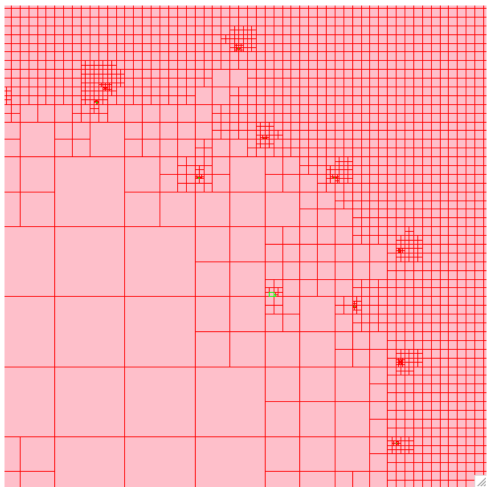

(b) The same figure zoomed to the top right.

Figure 3.1: Example visualisations.

`OpenGL` primarily because its transformation stack makes it easy for us to implement methods for zooming into the subdivision tree and panning it across our viewport. Our framework is in use across Core for the visualisation of other subdivision based methods.

3.4 The Newton type operators

3.4.1 The subdivision algorithm

Recall the generic subdivision algorithm presented in Section 2.1.2. We noted in Section 2.2.9 that for each of the Newton type operators $N(X), K(X), H(X)$, a zero in X will belong to the operator result as well. This allows us to use a slightly modified subdivision algorithm. It has an inner loop which is given by:

```
1.While $Q$ is non-empty
2.    Remove $B$ from $Q$.
3.    If $C_{out}(B)$ holds, discard $B$.
4.    Else
5.        $R \leftarrow P(B)$
6.        If $R \cap B = \emptyset$, discard $B$.
7.        Else if $R \subset B$
8.            Insert $B$ into the output.
9.        Else
10.           Subdivide $B \cap P(B)$ and insert the subdivisions into $Q$.
```

In the algorithm above, $P(B)$ can be any of the Newton type operators that we discussed earlier. Notice the change in line 10. Instead of subdividing B, we can subdivide $B \cap P(B)$ in its place. This does not affect the correctness of the algorithm but improves its efficiency as we can potentially discard areas of B that we know will not contain roots.

The general structure of the problem suggests that the code that implements the subdivision algorithm must be as loosely coupled with the code that implements the predicates $C_{out}(B), P(B)$ as possible. The algorithm needs to have no knowledge of how these predicates work, only requiring that they return the correct values as per their contract.

The bulk of our implementation is spread across three classes, the `Algorithm` class, the `BasePredicate` class and the `BoxT` class, each of which we describe below.

BoxT: The `BoxT` class represents a box in the 2D plane, as defined by ¶2. It is a `C++` template parametrised with `NT`, the number type of each of the boxes corners. The `BoxT` class contains instances of the `IntervalT` class that define its extent along the x and y axes.

In addition, the `BoxT` class implements a subdivide method. This splits the box into four equal boxes, each of which are $\frac{1}{4}$ the area of the original box. Recall that this method will be called when the box can neither be included nor excluded by our predicates.

Initial Box: Recall that the subdivision algorithm from Section 2.1.2 starts with an initial box B_0 which is an input to the algorithm. In our implementation, we either allow the user to specify the initial box in terms of the coordinates of its corners or estimate it using the input polynomial. To estimate it automatically, we assume that the user is interested in isolating all roots of the polynomial and choose a box that will contain all of them. The problem of choosing such a box is related to the theory of **root bounds** i.e., expressing upper and lower bounds on the roots of a polynomial in terms of its coefficients. In our work, we use an implementation that Core provides of the Cauchy Bound. The theory of these bounds is beyond the scope of this thesis and can be found in [43].

Box Sizes: In Section 2.1.2 we briefly spoke about practical implementation issues. We continue this discussion in the context of our implementation. At Level 1, as boxes get smaller we will start losing precision in the representation of their corners as boxes get smaller. Level 2 `BigFloats` do not suffer from similar issues as their corners are dyadic and will continue to be exact (though each of their representations will take larger amounts of memory). Further, as the depth of the subdivision tree grows the number of boxes increase considerably and may occupy a sizable fraction of the available system memory.

For this reason, the algorithm is run with a parameter denoting the minimum box size. Boxes that are smaller than the minimum box size are output as **unresolved**. We allow the minimum box size to be specified in two ways, either as a lower bound on the width of the box or as an upper bound on the depth of the box from the root of the subdivision tree.

BasePredicate: The `BasePredicate` class is the base class for all the predicates used in the subdivision algorithm. It contains some support methods that are common to all our predicates, but arguably its most important job is to define the API that the predicate implementations must implement. It does so by declaring some of its functions purely virtual.

```
enum OPERATOR_DECISION {
  OD_EXCLUDE = 0,
  OD_INCLUDE,
  OD_SUBDIVIDE
};

template <typename NT> class BasePredicate {
  // Returns true iff. the box is too small to be processed.
  // If this function returns true, the box is marked as
  // unresolved.
  bool Min(const BoxT<NT> *region) const;

  // Returns true iff the box is guaranteed to contain
  // no roots. This is guaranteed to be called before
  // Include, for a given box.
  virtual bool Exclude(const BoxT<NT> *region) const = 0;

  // Returns the operator decision for the box. Also appends
  // the results of P(B) to contraction.
  virtual OPERATOR_DECISION Include(
      const BoxT<NT> *region,
      vector<const BoxT<NT> *> *contraction) const = 0;
};
```

Note that this (abstract) class defines two functions, `Include` and `Exclude`. The algorithm implementation guarantees that for a given box, `Exclude` will be called before `Include`, and this is the equivalent of C_{out}. We then have `Include`, which can return the values defined by the enumerated type `OPERATOR_DECISION`.

Note that `Include` appends $P(B)$ to an input vector, instead of returning a box (note that its third argument is a pointer to a `vector<const Box*>` type). This is to deal with the case when extended interval arithmetic is used, where the operator output can be multiple disjoint boxes. Recall from Section 2.2.1, that division by an interval straddling zero in extended interval arithmetic might result in two disjoint intervals.

Algorithm: We are now ready to describe the inner loop of the algorithm. It is a direct translation of the pseudocode presented in Section 2.1.2. The algorithm operates on an initial `BoxT` with dyadic corners that represents our initial region of interest, and uses the predicates supplied by one of the subclasses of `BasePredicate` to operate on the box.

Recall that the generic subdivision algorithm specifies Q, a data structure that contains a "queue" of boxes that are in line to be processed by our algorithm. The `C++` standard template library (STL) provides two container classes that could be used in this situation, the `list<T>` container, and the `vector<T>` container.

While the former is backed by a (doubly) linked list, the latter uses an array. When a vector expands beyond its current capacity, it doubles its size by trying to allocate a larger chunk of contiguous memory. Inserts to the front and back of a `list`, as well as the removal of elements whose locations are known are constant time operations. The `vector` on the other hand allows for (amortised) constant time appends to its end, but linear time removals and additions from any position other than the end.

In our work, we choose to use a `vector` to implement a stack like data structure. New boxes are appended to the end of the `vector`, and boxes are removed from the end of the vector for processing. Both of these operations are constant time operations. A `vector` used in this way proves to be faster than a `list`, possibly due to better memory locality. Further, it tends to[8] use less memory than a `list` which has a constant overhead for every element it stores (due to linked list pointers).

A consequence of this decision is that the subdivision of our search set tends to be "depth first". Results of a subdivision are appended to the end of the `vector` and all of them will necessarily be processed ahead of other boxes in the `vector` since we remove boxes from the end of the vector for processing.

3.4.2 The exclusion predicate

Recall from Lemma 1 that our exclusion predicate has a very simple form:

$$0 \notin \square u(X, Y) \textbf{ or } 0 \notin \square v(X, Y).$$

All it requires is the evaluation of the interval extensions $\square u$ and $\square v$. We mentioned in Section 2.2.2 that in the case of polynomials, the interval extension can be obtained by replacing the real number operations $\{+, -, \cdot\}$ with their interval equivalents. The naive implementation can proceed as per Section 3.3.3 by using the existing polynomial evaluation code instantiated with the `IntervalT` type. Since we use the same exclusion predicate along with each of the three operators, it is implemented in the `BasePredicate` class (the `Exclude` function).

Note that since polynomial evaluation (using Horner's rule or otherwise) can be reduced to the operations $\{+, -, \cdot\}$, it is in general exact at Level 2 when evaluated at a dyadic point with polynomial coefficients that are dyadic.

For polynomials of a higher degree, the interval extensions' estimate of the range of u, v or a box B can be much wider than the actual range (see 4.2.1). Converting the polynomials into a centred form centred at $m(B)$ can yield a much tighter estimate of the polynomial range. (See [41] for an exposition of the effect of the polynomial representation on the interval extension) This improvement in accuracy of due to the centred form is balanced to some extent by the cost of recentring the polynomial at the midpoint of every box.

The centred form can be obtained either by calculating the coefficients of the bivariate Taylor series at the new centre, or by composing the polynomials with the linear transformations $x+u = x_c$ and $y+v = y_c$ where x_c, y_c are the new centres. With this transformation, a bivariate polynomial $f(x, y)$ is transformed into $g(u, v)$ say. Then, g can be evaluated at $(x_c - x, y_c - y)$, which has the same effect as a centred form.

We seek to balance the cost of recentring by using a simplistic scheme of basing the decision to recentre the polynomial on a threshold T. Here T is the number of subdivisions since the last centring. Note that the number of subdivisions is a useful heuristic for this purpose because a subdivision occurs only if both the inclusion and exclusion predicates do not hold. If we find that both predicates do not hold more often than they do, it might be an indicator that they are not very effective. A detailed tabulation of run times and other measures of efficacy with and without these changes can be found in Section 4.2.1.

[8]Note that we say "tends to", note that a vector that has just doubled its size has slightly less than half of its capacity unused. In some pathological cases, this might lead to memory pressure.

3.4.3 The inclusion predicates

In this section we describe the three subclasses of the `BasePredicate` class that implement each of the Newton, Krawczyk and the Hansen-Sengupta inclusion tests.

The Newton operator: The Newton operator is given by (2.16) and is

$$N(\mathbf{X}) = \mathbf{x}_c - \square J(\mathbf{x})^{-1} \cdot \mathbf{f}(\mathbf{x}_c).$$

We deal here with the two dimensional case, and so $\mathbf{X} = (X, Y)$ where X, Y are the x and y axis extents of a given box B. Further, $\mathbf{f} = (u(x, y), v(x, y))$ and $\mathbf{x}_c = (m(X), m(Y))$. The Jacobian of this two dimensional system is given by

$$J(u, v) = \begin{bmatrix} u_x & u_y \\ v_x & v_y \end{bmatrix}, \tag{3.1}$$

and its interval extension is naturally

$$\square J(u, v) = \begin{bmatrix} \square u_x & \square u_y \\ \square v_x & \square v_y \end{bmatrix}.$$

The matrix inverse in this case is straightforward, and is

$$\square J(u, v)^{-1} = \frac{1}{D} \begin{bmatrix} \square v_y & -\square u_y \\ -\square v_x & \square u_x \end{bmatrix}, \qquad D = (\square u_x \cdot \square v_y - \square u_y \cdot \square v_x). \tag{3.2}$$

With these definitions in hand, the translation of the Newton operator into code is straightforward. We use the Core linear algebra package [9] to represent our two dimensional vectors and the interval matrix $\square J(\mathbf{X})^{-1}$. Once this is done, the code mirrors equation (2.16) exactly.

Note that if a box B is dyadic, then its x and y extents (X, Y) will have dyadic endpoints and their centres $(m(X), m(Y))$ will be dyadic as well. This means that all multiplications and additions involving intervals will be exact at Level 2, and as a consequence as will all polynomial evaluations. The only operation that may not be exact is the division $\frac{1}{D}$ in (3.2), but here we round the resulting interval endpoints correctly as described in Section 3.3.3. The resulting $N(X)$ might hence be wider than if it had been calculated at infinite precision. Obviously, this does not affect the correctness of the operator in any way. Note that if we rounded our end points incorrectly, the result interval might have been *narrower* than the exact result, and the test might have passed even though it should not.

Note that the determinant term D in (3.2) might be an interval containing a zero. There are two ways in which this can be dealt with.

- **Variant 1**: Use the rules of extended interval arithmetic (See Section 2.2.1) to compute this quotient.
- **Variant 2**: Subdivide the box by splitting it into four equal sized boxes.

We present a comparison of the two approaches in Section 4.2.2. Note that if D contains a zero, the box B will never be isolating since the results of extended division are always infinite. However, its usage might help us bound the root away from certain regions of the box (since if B contains roots of f, so will $B \cap N(B)$).

As an example, we run our algorithm on the polynomial $z^{10} - 1 = 0$. The roots of this polynomial are the 10^{th} roots of unity and are equispaced on the unit disk. The command line arguments are given below to illustrate how our program can be used.

```
./main_newt --display                \\ Display subdivision tree
            --poly data/nroots10.pol \\ Input polynomial file
            --use_root_bounds
            -n                       \\ Use the newton operator.
```

Note the `use_root_bounds` command line argument that tells the program to estimate the input box B_0. In this case, the initial box is $B_0 = [-2, 2] \times [-2, 2]$. The subdivision tree that this produces can be seen in Fig. 3.2. Note that areas near the roots are subdivided further than areas that are further away from the roots.

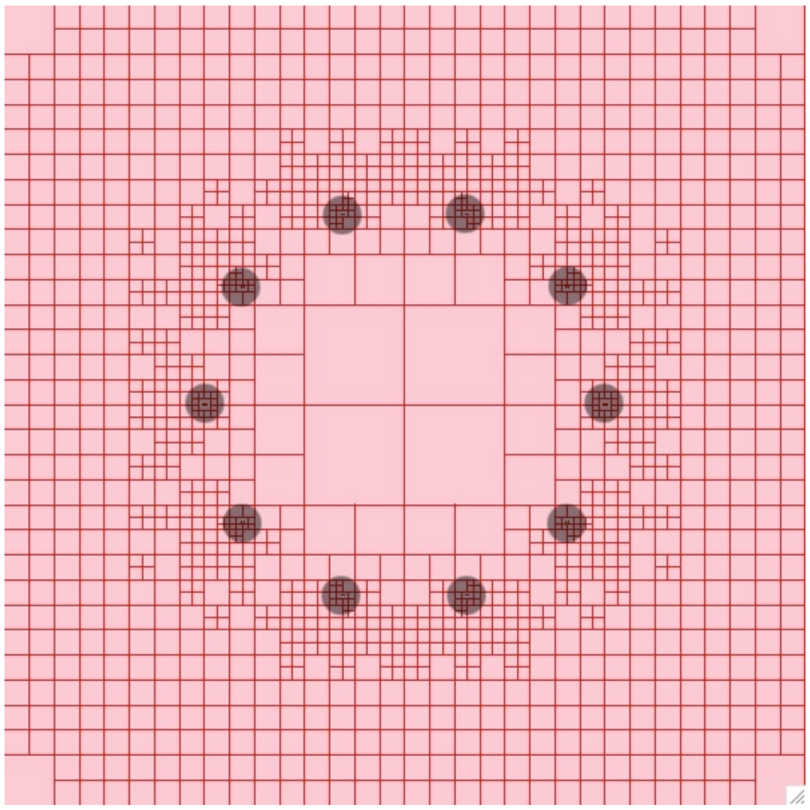


Figure 3.2: Subdivision tree for $z^{10} - 1 = 0$ using the Newton operator

The Krawczyk operator: The Krawczyk operator is defined in (2.21) and is given by:

$$K(\mathbf{X}) := \mathbf{y} - Y \cdot \mathbf{f}(\mathbf{y}) + \{I - Y \cdot \square J(\mathbf{X})\}(\mathbf{X} - \mathbf{y}).$$

Here $\mathbf{y}$ is taken to be $(m(B_x), m(B_y))$ where B_x, B_y are intervals corresponding to the x and y extents of B respectively. Also, X, $\mathbf{f}$ and $\square J(\mathbf{X})$ have the same definition as with the Newton operator and Y can be any non singular matrix.

We have three strategies for calculating Y, all we require is that it is non singular:

- **Variant 1**: Choose $Y = J^{-1}(\mathbf{y})$ i.e., the inverse of the Jacobian evaluated at the centre of B.
- **Variant 2**: Choose $Y = \text{mid}(\square J(\mathbf{X})^{-1})$ [9].

[9]The midpoint of an interval valued matrix A is simply the matrix whose elements are the mid points of the elements of A

- **Variant 3**: Choose Y as some fixed non-singular matrix, for example $I_{2\times 2}$.

In the case of variants 1 and 2, irrespective of Core Level, the inverse can be calculated at fixed (machine) precision. We do not care about exactness because Y can be **any** non singular matrix.

Further, note that our expressions for Y might yield singular matrices. In this case we can always fall back to using a fixed non singular Y. The operator is expected to be sensitive to the choice of Y since $w(K(\mathbf{X}))$ depends on that of $\{I - Y \cdot \square J(\mathbf{X})\}$. We provide a comparison of the efficiency of these variants in Section 4.2.3.

As in the case of the Newton operator if the box B has dyadic corners the evaluation of $K(X)$ will be exact, or rounded correctly.

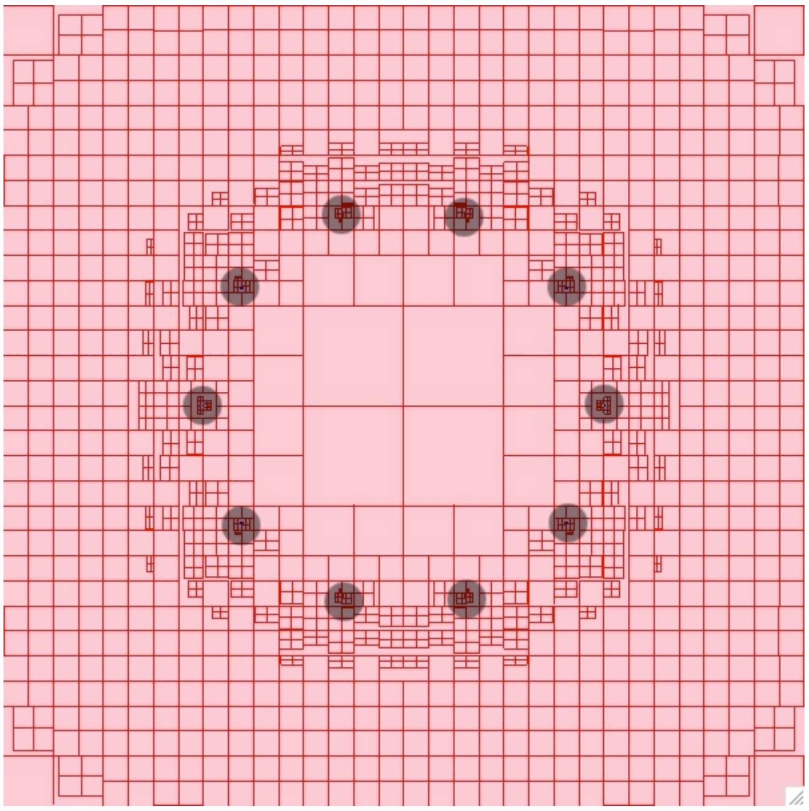


Figure 3.3: Subdivision tree for $z^{10} - 1 = 0$ using the Krawczyk operator

The Hansen-Sengupta operator: The Hansen-Sengupta operator is defined by equation (2.25) and is:

$$\mathbf{H} = \mathbf{x} - D^{-1}\left\{\mathbf{l} + L(\mathbf{X}' - \mathbf{x}) + U(\mathbf{X} - \mathbf{x}),\right\} \qquad \mathbf{X}' = \mathbf{H} \cap \mathbf{X}$$

where $U + D + L = A$, $A = B \cdot \square J(\mathbf{X})$ and $\mathbf{l} = -B \cdot \mathbf{f}(\mathbf{x})$ for a suitable preconditioner matrix B. In our implementation, we choose $B = J(\mathbf{x})^{-1}$ though the choices are the same as in the case of Y in the Krawczyk operator.

In the two dimensional case, equation (2.25) becomes considerably simpler. If $\mathbf{X} = (X, Y)$, and $\mathbf{x} = (x, y) = (m(X), m(Y))$ then we can calculate the first element of $\mathbf{H}$ above as:

$$X_1 = x + (l_1 - A_{12}(Y - y))/D_{11}. \tag{3.3}$$

We then calculate $X' = X_1 \cap X$ to use in our calculation Y_1 which is:

$$Y_1 = y + (l_2 - A_{21}(X' - x))/D_{22}. \tag{3.4}$$

Now, $\mathbf{H} = (X_1, Y_1)$ is the result of the operator on a box B with extents (X, Y). If either of D_{11} or D_{22} is zero, we must resort to extended interval arithmetic in the calculation of this quotient. Even if X_1 is of infinite length as a result of an (extended) division by zero, $X_1 \cap X$ will be finite. Further X_1 might be a union of up to two intervals if a divide by zero occurs and so might Y_1. What this implies for a box B is that $H(B) \cap B$ might the union of four disjoint boxes. It is important to stress that if this happens, B is never an isolating box as $H(B)$ will always be infinite. However, $H(B) \cap B$ can give us information as to where in B the roots **cannot** be.

As in the case of the operators above, a dyadic box B will yield dyadic, or correctly rounded results.

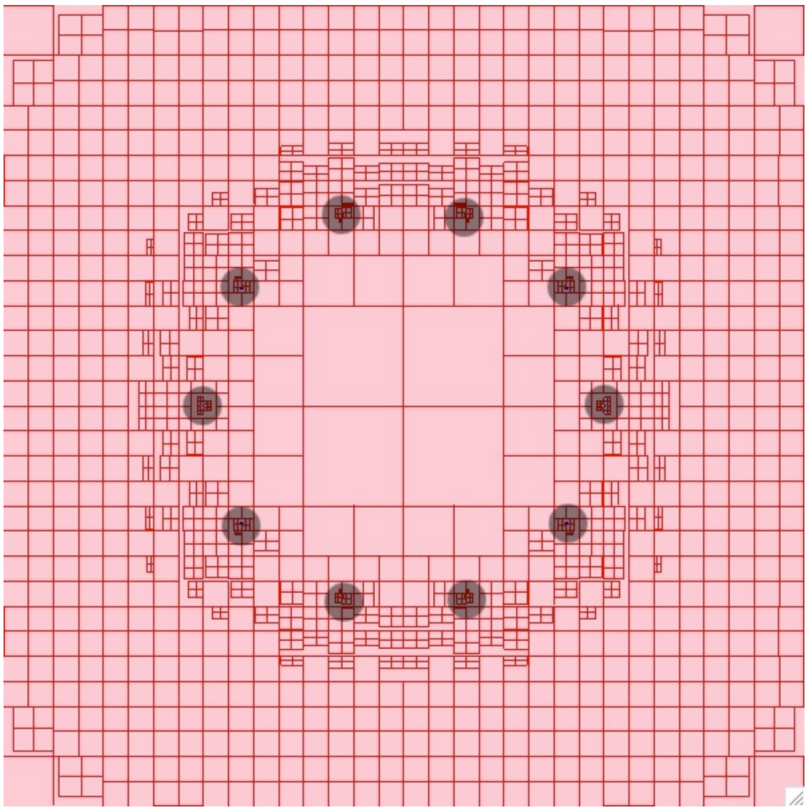


Figure 3.4: Subdivision tree for $z^{10} - 1 = 0$ using the Hansen-Sengupta operator

3.4.4 Dealing with unresolved boxes

In Section 2.2.6, we spoke about the case where a root that is at the boundary of a box will result in the algorithm terminating with unresolved boxes. Another cause for unresolved boxes is the overestimation of function ranges by interval analytic predicates (see Section 4.2.5).

We deal with unresolved boxes as per Section 2.2.6. We now take the example of $z^3 + z = 0$, which has the roots $\{0, \mathbf{i}, -\mathbf{i}\}$. Trying to resolve its roots with $B_0 = [-2, 2] \times [-2, 2]$ should lead to unresolved boxes around the roots which lie on the boundary of boxes at the centre of the region. These boxes have x extents of the form $[a, 0]$ and $[0, b]$. The output of running of running the algorithm without a further step to deal with unresolved boxes is shown below.

```
$ ./main_newt --poly ./data/boundary.pol --display -n
iters=325,includes=0,splits=81,ambiguous=48,exc_c0=196,exc_c1=0,time=19

Operating over : [ -2, 2 ],[ -2, 2 ]
With polynomials :

[x + x^3] + [-3x] * y^2
[1 + 3*x^2] * y^1
 + [-1] * y^3
Output regions :

Unresolved regions :
X : [ -5.1280538749490776375e-05, 0 ], Y : [ 1, 1.0000166586618850495 ]
X : [ -0.00032521731005719642978, 0 ], Y : [ 0.9994359129023973054, 1 ]
<truncated>
```

Note that there are 48 unresolved regions, the full list of which has been truncated. Further, note that the program found no roots. All of the roots are contained within the unresolved regions in the output. This is shown in Figure 3.5, where regions of blue are unresolved boxes. These are too small to see without sufficient magnification, so we have highlighted them. We

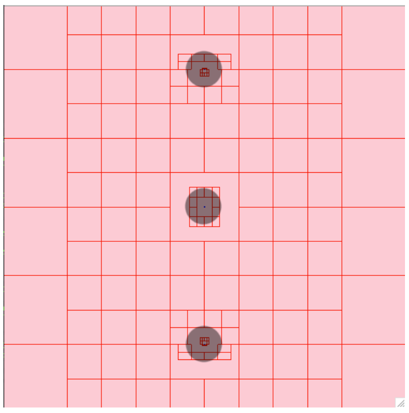


Figure 3.5: Subdivision tree for $z(z^2 + 1) = 0$ using the Newton operator

now list the output of the program when we run an additional step to deal with unresolved boxes (note the additional command line parameter `--step_2`). The output shows that all three roots have now been isolated.

```
$ ./main_newt --poly ./data/boundary.pol --display -n --step_2
iters=325,includes=0,splits=81,ambiguous=48,exc_c0=196,exc_c1=0,time=28
Operating over : [ -2, 2 ],[ -2, 2 ]
With polynomials :

[x + x^3] + [-3x] * y^2
[1 + 3*x^2] * y^1
 + [-1] * y^3

Output regions :
X : [ -9.5128978964329236225e-06, 9.5128978964329236225e-06 ],
     Y : [ 0.99997358089918875201, 1.0000288677637101387 ]
X : [ 0, 0 ], Y : [ 0, 0 ]
X : [ -9.5128978964329236225e-06, 9.5128978964329236225e-06 ],
     Y : [ -1.0000288677637101387, -0.99997358089918875201 ]

Unresolved regions :
```

3.5 The CEVAL algorithm

Our discussion of the CEVAL algorithm is largely limited to the implementation of T_K and the 8-Point-Test which serve as its exclusion and inclusion predicates respectively. We start with a definition and an observation.

Representing a complex number: As described in Section 3.3.4, the machine representation of a complex number is a pair of real numbers. The type of these numbers depends on the Core Level. At Level 1, the real and imaginary parts are fixed precision floating point numbers, and suffer from the same precision issues as their real counterparts.

At Level 2, the real and imaginary parts are `BigFloat` numbers and can store dyadic real and imaginary components exactly. We abuse terminology slightly and call a complex number with dyadic real and imaginary parts a *dyadic complex number.*

The addition and subtraction of two dyadic complex numbers is obviously dyadic. The product of two dyadic complex numbers $z_1 = a + \mathbf{i}b$, $z_2 = c + \mathbf{i}d$ is computed as $z_1 \cdot z_2 = (ac - bd) + \mathbf{i}(bc + ad)$ and is also dyadic since its real and imaginary portions are built from dyadic real numbers using the operations $\{+, \cdot, -\}$. The quotient of two complex numbers $z1, z2$ is:

$$\frac{z_1}{z_2} = \frac{a + \mathbf{i}b}{c + \mathbf{i}d} = \frac{ac + bd}{c^2 + d^2} + \mathbf{i}\frac{bc - ad}{c^2 + d^2}.$$

Clearly, this is not necessarily dyadic as the real and imaginary parts are divided by the factor $(c^2 + d^2)$. As a final point, the modulus of a complex number $|z|$ need not be dyadic due to the presence of a square root.

The subdivision algorithm: Unlike the Newton type operators, the subdivision step of this algorithm always divides a box into four equally sized children, and we do not substitute B with $P(B) \cap B$ before subdivision (such a substitution has no meaning in this case). Apart from this difference, the algorithm is exactly the same as that for the Newton type operators and we do not discuss it any further.

One point to note is that if a box B with centre m and radius r passes the 8-point-test, we insert $D(m, 4r)$ into the output. From Section 2.3.3, we know that this disk D might intersect other disks in the output, in which case we can resolve this by including the smaller of the two disks and discarding the other. The output list is an STL `list` container, and we search it linearly for intersecting disks replacing them in place. Note that number of disks in this list is bounded by the degree of the polynomial under consideration, and is never prohibitively large.

3.5.1 The T_K predicate

Recall the definition of the T_K predicate from equations (2.28) and (2.29) given by:

$$T_K^f(m, r): \quad |f(m)| > K \sum_{k \geq 1} \left| \frac{f^{(k)}(m)}{k!} \right| r^k, \tag{3.5}$$

and the closely related version for the derivative f':

$$T_K^{f'}(m, r): \quad |f'(m)| > K \sum_{k \geq 2} \left| \frac{f^{(k)}(m)}{(k-1)!} \right| r^{k-1}. \tag{3.6}$$

Note that we cannot guarantee that both sides of the inequality will be exact due to the presence of the modulus and division on both sides of the equation. Consider $T_K(B)$ applied on a box B.

Making K dyadic: As part of the CEVAL algorithm, K can be one of $\{1, \sqrt{2}, 6\}$ where T_1 forms the exclusion predicate and $T'_6, T'_{\sqrt{2}}$ are preconditions for the 8 point test. Clearly, we would like K to be dyadic, as it appears in the RHS of (3.5) and (3.6) above. We can replace $\sqrt{2}$ with $\frac{3}{2}$ [37] since T_a implies T_b if $a, b > 0$ and $a > b$. This is a consequence of the RHS of the equations for (3.5),(3.6) strictly increasing with increasing K.

Making r dyadic: Note that the radius of a box B defined by equation (2.1) is not in general dyadic. Like in the above case, the RHS of (3.5) strictly increases with increasing r, so $T_K(m, r_1)$ implies $T_K(m, r_2)$ for $r_1, r_2 \in \mathbb{R}+$ and $r_1 > r_2$. We can therefore replace $r(B)$ with an approximation to it, the most obvious of which is $r(B)$ rounded towards $+\infty$. Note that [37] suggests that $\frac{3}{4}w(B)$ can be used as well, based on the same justification.

Issues with modulus, division: The issue with the division by $k!$ can be solved by multiplying both sides of the inequality with $n!$ where n is the degree of f, and thereby verifying that

$$n!|f(m)| > \sum_{k\geq 1} n.(n-1)\ldots(n-(k-1))|f^{(k)}(m)|r^k.$$

The last observation is that if all our $|z|$ on the LHS are rounded towards $-\infty$ and those on the RHS are rounded towards $+\infty$, the correctness of the predicate is not affected.

Issues with polynomials of higher degree: Note that the value of $n!$ increases very rapidly as the degree n of the polynomial increases. In fact, 22! is the largest factorial that can be represented by a 64 bit integral type. Even if we are working at a Core level that has an extended precision integer type, the number of bits required to represent an integral factorial value can be large enough to slow down the entire algorithm. We therefore take the approach of using a floating point type (rounded in the correct direction) to store approximations of the factorial to a fixed precision. Consider the predicate:

$$n!|f(m)| > \sum_{k\geq 1} n.(n-1)\ldots(n-(k-1))|f^{(k)}(m)|r^k.$$

Now clearly, if the $n!$ term on the LHS is rounded downwards (towards $-\infty$) and the $\frac{n!}{k!}$ terms on the RHS are rounded upwards, when the predicate holds at a finite precision it will hold at infinite precision as well. We can compute all of these terms once per run of the algorithm, and store them in a table for quick lookup. Another option might be to use a known mathematical approximation for $n!$, such as Stirling's approximation, which is:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n.$$

This might be a viable alternative for very large n, but it is one that we have not explored yet.

Note that the changes listed above make the predicate T_K more conservative, as we end up evaluating it at marginally larger box radii (rounded upwards) and K values (also rounded upwards). This suggests that at finite precision, T_K might fail when it reality it should hold. This does not affect the correctness of the overall algorithm, because if these predicates fail, a box is never (wrongly) rejected, rather it is processed in more detail after being subdivided.

3.5.2 The 8-Point-Test

Assume that $T'_6(m, 4r)$ and $T'_{\sqrt{2}}(m, 8r)$ hold. Then, the 8-point test requires us to evaluate the sign of u, v at eight points on the boundary of $D(m, 4r)$. These points are the end points of eight arcs, and are given by:

$$P = \left\{ m + r \cdot e^{\mathbf{i}j\pi/4} \right\}, \qquad j = 0, 1, \ldots, 7.$$

Using $e^{i\theta} = \cos\theta + \mathbf{i}\sin\theta$, we can arrive at expressions for the 8 points at which we must evaluate f. The cardinal compass points $\{N, S, E, W\}$ are in general dyadic, as they amount to $\{m \pm r, m \pm \mathbf{i}r\}$ for a dyadic m,r. The other 4 compass points are $\left\{ m + \frac{r}{\sqrt{2}}(\pm 1 \pm \mathbf{i}) \right\}$ and are not dyadic and hence the evaluation of f can never be exact at these points.

To evaluate the exact sign of f (and thereby that of u, v), we follow the recommendations of Section 6 of [37]. They note from Theorem 14 of [37], that it suffices if each pair i, j of compass points are separated by an angle θ_{ij} where $\theta_{ij} \in [45° \pm \delta]$, and $\delta = 2.5°$. We can now try to find a Pythagorean triplet (a, b, c) such that $\sin\theta = \frac{a}{c}$ and $\cos\theta = \frac{b}{c}$ are representable by the rational number `BigRat` type. In this situation, r which is dyadic and therefore rational, can also be represented as a `BigRat`. This lends each of these points to an exact rational representation (with dyadic numerators and denominators), and therefore the polynomial can be evaluated exactly in this form.

Note that even though we need the signs of u, v at these points, these are evaluated by calculating $z = f(p)$ and taking the signs of `Re`(z) and `Im`(z).

3.6 The CXY algorithm

This section details some of our contributions to the CXY algorithm [25] for topologically accurate meshing of implicit curves. The CXY algorithm is not related to the rest of this thesis, except that it uses interval arithmetic and is a subdivision algorithm. The reader may therefore skip this section without any loss of continuity. Note that this section assumes familiarity with the CXY algorithm and related terminology from [25].

3.6.1 Contributions to CXY

Most of our changes to CXY are the result of careful CPU and memory profiling of its execution. The other changes are due to stronger invariants that we could impose on its data structures that were afforded to us by theory in [25]. Some of our modifications are summarised below.

- Profiling revealed that comparing widths of boxes was a time consuming operation. We observed that there was no need to compare the actual width of a box, and all boxes at the same depth from the root of the subdivision tree have the same width. Using this, we could translate all box size implementations into unsigned integer arithmetic.

- We note from Section 5 of [25] that we do not require a "total" priority queue over all boxes. All we require is that smaller boxes appear ahead of larger boxes, and boxes of the same size can be processed in any order except for the requirement that ambiguous boxes appear ahead of all others. This lends itself to an implementation that maintains a list of lists of boxes, each of these lists containing boxes of a particular size. The number of such lists is the same as the depth of the subdivision tree and in general is small. Such an implementation will provide worst case time complexities of $O(1)$ on the `Insert` and `Erase` operations respectively.

- We now ensure that the balancing step only balances in-boxes, and not every node in the subdivision tree.
- We implemented an efficient balancing algorithm that propagates changes from the minimum sized boxes outwards. It ensures that the minimum possible number of boxes are compared and split. Further, it brings with it support for balancing the subdivision tree around a particular node. In the previous implementation, all balances were global.

3.6.2 Results

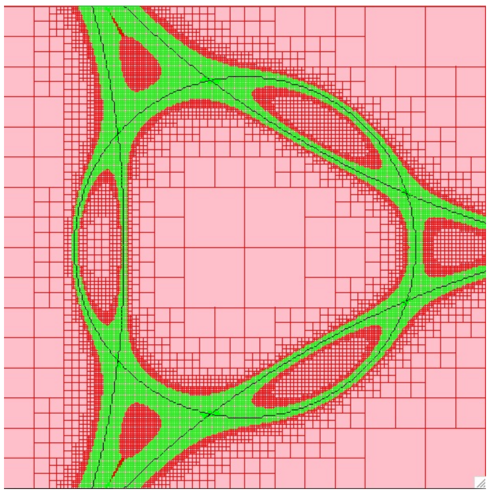
The new code is much more efficient than the original, and since it occupies a smaller memory footprint, can handle much larger subdivision trees. Further, the use of our new interval arithmetic package and improved polynomial evaluation result in a speedup in the subdivision step of the algorithm. The newly written balancing code is probably the most dramatic improvement, yielding a 10x improvement in run time on an average. Results of run times on some functions are listed below. Each of these targets correspond to the CXY makefile in Core. In all cases,

Target	Old	New	Speedup
eg1	8 ms	6 ms	1.3x
eg4	202 ms	70 ms	2.9x
eg12	775 ms	224 ms	3.5x
eg31	13,723 ms	805 ms	17.0x
eg33	179,472 ms	6183 ms	29.0x

Table 3.1: Comparison of CXY running times

the new code ran at least as fast as the old and in most situations was dramatically faster. Our new implementation will serve as the base for the extension of CXY to higher dimensions and its extension to calculating curve intersections. The results of meshing some of the curves above are shown below. The two curves are

$$\begin{cases} \text{eg30} & : (x^2+y^2+12x+9)^2 - 4(2x+3)^3 = 0 \\ \text{eg33} & : (x^3+y^3-3xy)(x^2+y^2-0.1)((x^2+y^2)(y^2+x(x+1)) - 4xy^2) = 0 \end{cases}$$


(a) eg30

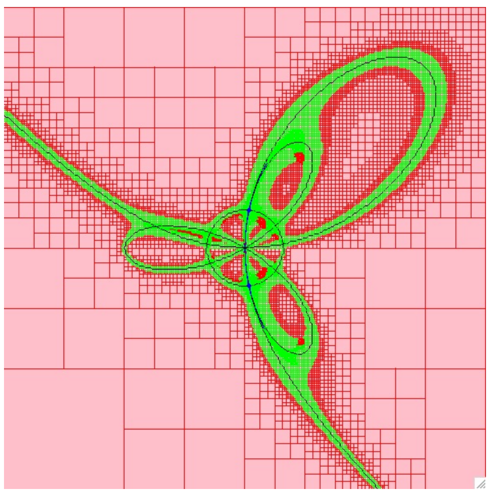

(b) eg33

Figure 3.6: Example meshes constructed by CXY.

Chapter 4

Results

In this chapter, we discuss the performance of the operators we implemented as per Chapter 3. Section 4.1 describes the setup and methodology that we used for our tests, and we deal with our test results in Sections 4.2 and 4.3.

4.1 Test setup

4.1.1 Testbed details

All of our code is compiled into two separate binary executables (programs). The first of these programs isolates roots as per the interval arithmetic based methods, and here the choice of operator is controlled by a command line argument. The CEVAL algorithm shares very little common code with the above and is compiled into a separate binary. The inputs to these programs are provided in the form of command line arguments, a full listing of which can be found in Appendix B. (For sample usage, see Section 3.4.3, Appendix C).

To generate tabular results, these programs are run by a Python script using the appropriate command line argument combinations. For this purpose, we use the Python `subprocess` module to execute a command and to collect and process the output it writes to the standard terminal. In this way, the python script can run the program over a series of inputs and generate tabular data that can be directly included into LaTeX.

Both of these programs support two output formats. The first of these is a detailed format that includes the subdivision diagrams of the previous chapter along with various diagnostic information and is meant for human consumption. The second is a compact mode that produces output that is meant to be parsed by a script or machine. An example of the latter is given below.

```
$./main_newt --random \
             --seed 20130 \
             --degree 15 \
             --use_root_bounds \
             -n
iters=4525,includes=12,splits=1131,ambiguous=48,exc_c0=3326,exc_c1=8,time=4189
```

The output is a comma separated list of key value pairs. A description of what these keys represent can be found in Section 4.1.3.

Hardware: All tests were run on a MacBook Pro with an Intel Core 2 Duo processor clocked at 2.53 GHz with 4GB of RAM running `Mac OS X (10.5.8)`. All compilation was performed using `gcc-4.2` with its most aggressive optimisation flags (`-O3`).

Core Level: All tests in this section can be assumed to have run at Level 1 except where indicated. Recall that Level 1 operates at fixed precision, and calculations at this level are not in general exact for most dyadic numbers.

To counter this, our algorithms are run with the `min_box_size` parameter set sufficiently high to mitigate some of these issues. Moreover, our input polynomials are such that machine precision is sufficient to isolate their zeroes. In each case, we have checked the correctness of our Level 1 output against other polynomial solvers (primarily MPSolve), and also with our own code running at a higher precision.

We also note that the *relative* performance of these operators remains the same as we operate at higher Core levels, except that the differences between them are more pronounced than at Level 1. We provide a comparison of run times at Level 2 in section 4.3.4, and in Section 4.3.5 we discuss an where the attempt to isolate roots at Level 1 fails, and operation at Level 2 is required.

4.1.2 Test Polynomials

Our test polynomial set consists of a few specific well known polynomials, along with a series of randomly generated polynomials of varying degrees. Our examples are drawn from the MPSolve test suite [5], which is itself a part of the FRISCO[1] test suite [40].

Specific polynomials: These are chosen from among polynomials that appear often in the areas of probability, differential equations and graph theory. Examples of such polynomials are the degree 20 Hermite, Chebyshev and Laguerre polynomials, and a Chromatic polynomial of degree 22. Appendix C contains a list of some of these polynomials and their coefficients, along with the program output for each of them. Other examples include:

- **n^{th} roots of unity:** These are the solutions of the equation $z^n - 1 = 0$ and are equispaced points on the unit disk, i.e., $|\alpha| = 1$ for each root α. We include $n = 10, 20$ as part of our standard tests.
- **Wilkinson's polynomial:** These are polynomials of the form $\prod_{k=1}^{n} (z - k)$. The roots of these polynomials are real and are simply $1, 2 \ldots n$. Note that the coefficients of these polynomials are in general large (The constant term will be 20! for the degree 20 polynomial).

Randomly generated polynomials: We implemented a routine in `C++` that generates polynomials with (pseudo) random coefficients. The routine requires two parameters, the maximum degree of the polynomial and an upper bound on the magnitude of its coefficients (which for all tests is 10 unless specified otherwise). This ensures that the roots of the polynomials fall within a reasonable bounding box.

Note that the seed for the random number generator is provided as an input to our program, so we can ensure that the same sequence of pseudo random polynomials is generated and processed across multiple runs of a given test.

Lastly, note that polynomials above, such as the Wilkinson and Chebyshev examples have much larger coefficients than our randomly generated polynomials.

[1]**Fr**amework for **I**ntegrated **S**ymbolic/Numeric **Co**mputation, a three year project funded by the European Commission

4.1.3 Statistics

For each run of the algorithm, we collect statistics that we consider indicative of the performance of the algorithm. The most important of these statistics is the algorithm running time:

Running time: The performance of the algorithm is measured in terms of wall clock time. This is measured by specific `C++` code that we implemented. We use the `C gettimeofday` API to measure microsecond start and end times and use them to calculate our running time. Note that wall clock time is not always a reliable indicator of running time due to factors like context switches between processes and so on. However, across multiple runs of a given program it is a reliable indicator of its relative efficiency. Moreover, our programs are CPU intensive, and do not perform any disk or network I/O operations and hence the running time measured in this way is a suitable metric.

In addition to the running time of the code, we collect other statistics such as those described below:

Number of boxes processed: We keep track of the total number of boxes processed during an algorithm run. Recall that each box is either included, excluded or subdivided. Obviously, this number will depend on the size of the initial box B_0 but for initial boxes of the same size, a method that processes fewer boxes is likely to run faster.

Number of C_{out} (resp. C_{in}) excludes: This is the number of boxes that were excluded by the C_{out} (resp. C_{in}) predicate. Recall that the inclusion predicate can reject boxes in the case of the Newton type operators when $P(B) \cap B = \emptyset$ where P is one of the operators. This number should be treated carefully and not viewed in isolation because smaller values are not necessarily indicative of poor performance. An example is the case when a larger box can be excluded instead of subdividing it into smaller boxes and then individually excluding each of the subdivisions.

Number of unresolved boxes: As we mentioned in Section 2.1.2, the algorithm is run with a parameter that specifies the maximum allowed extent of subdivision. Boxes that cannot be excluded or included at that point are marked as unresolved. Ideally, this number should be zero and unless it is so, we cannot guarantee that all roots contained in the initial box B_0 have been isolated.

4.2 Newton type operators

We now discuss the performance of each of the Newton type operators individually, and provide a comparison of their performance at the end of this section.

Result tables: In our tables, "**Iters**" stands for the total number of boxes processed during the running of the algorithm (**iter**ations of the main loop, see Section 2.1.2). Also, all of our result tables are sorted in increasing order of polynomial degree. When a given metric is being compared across multiple approaches, the best performer is always underlined.

4.2.1 The exclusion predicate

We start with a small note about the exclusion predicate. We noted in Section 3.4.2 that using a centred form in the evaluation of a polynomial over an interval yielded a tighter result. We now provide data to back our claim. Recall that T is the threshold number of failures at which we recentre the polynomial. At $T = 0$ we recentre at every application of the exclusion predicate, and as T gets higher, we recentre less frequently. We present Table 4.1, which contains run times for the interval Newton operator isolating the roots of $z^{20} - 1 = 0$.

T	Iters	C_{out} (%age of Iters)	Time(s)
0	8197	6032(73.5)	12.931
4	15257	11295(74.0)	1.736
16	31125	22084(70.9)	1.254
64	70777	44187(62.4)	1.493
256	182369	79776(43.7)	2.706
$+\infty$	3464677	81676(2.3)	17.4

Table 4.1: The effect of T on the exclusion predicate. Observe that the percentage of boxes that the C_{out} predicate can exclude gets lower as we recentre less frequently.

Note that the number of iterations steadily increases as we decrease the frequency of centring. The running time in contrast, decreases, as the computational effort of recentring need not be borne. As we increase the value of T beyond 64 however, the cost of processing a higher number of boxes negates any gains from less frequent centring, and the run time starts to increase again.

Also, note that this heuristic for recentring the polynomial is sensitive to the order in which we process boxes. If boxes that are "close" to each other are processed subsequently, we would expect that the centred form would need to be calculated less frequently.

We would like to emphasise that this is a rather simplistic scheme for improving predicate performance. There is scope for improvement, but we would like to note that range overestimation by the exclusion predicate seems to be an inherent drawback of the interval arithmetic based exclusion approach. We will see in Section 4.3.1 how the T_K predicate compares to this one.

Another variant of this predicate that we implemented uses the definition of exponentiation from Section 2.2.1 to try to obtain a tighter result. Recall that we have specific expressions for I^n in this case, and we do not evaluate it using $I^n = I \cdot I \ldots$. We switched from using the Horner's rule (where this change would have no effect as there is no explicit exponentiation) to a sum of powers based evaluation strategy in this case. Sample results of this approach can be found in Table 4.2. Comparing these with Table 4.1 which contains data for the same polynomial shows that except for $T = 0$, the Horner's rule performs better. *We therefore choose to use the Horner's rule and the standard definition of exponentiation for the rest of our experiments.*

T	Iters	C_{out}	Time(s)
0	7253	5324	12.305
4	22877	16995	3.734
16	48649	33708	4.043
64	121681	69184	7.172
256	305145	108671	12.269

Table 4.2: The effect of using power summation in the exclusion predicate. Observe that the number of iterations is higher than Table 4.1 except for $T = 0$.

4.2.2 The interval Newton operator

In Section 3.4.3, we discussed two variants of the interval Newton operator. The first of these variants (Variant 1) used extended interval arithmetic when a singular Jacobian was encountered and the second chose to subdivide the box in this case (Variant 2). We now present results of tests that exercise the two variants.

All tests in this section are run over the input box $B_0 = [-2, 2] \times [-2, 2]$ with the aim of isolating all roots within that box. In some cases, the input polynomial may not have a root within B_0, but even so this test will give us an indication of how good the algorithm is at discarding regions that have no roots. Further, we do not run the additional step to resolve unresolved boxes as per Section 3.4.4. The number of unresolved boxes is not reported here as it is the same across variants (this data can be found in Table 4.5).

We run the variants on our standard test set of polynomials described in Section 4.1.2. In the case of the randomly generated polynomials, we generated 50 random polynomials of each of the degrees $n = \{4, 6, 8, 10, 12\}$ and each table entry contains the mean of the statistics for that degree.

	Iters		Time(s)		Num zero divs.
	v1	v2	v1	v2	
random4	348	331	0.032	0.027	49
random6	788	766	0.1356	0.125	49
random8	1385	1336	0.415	0.368	349
random10	1958	1936	0.875	0.823	475
nroots10	2245	2213	1.025	0.885	517
random12	2861	2818	1.806	1.699	759
chebyshev20	13717	13573	22.928	21.955	3321
nroots20	8341	8245	13.723	12.687	1997
laguerre20	869	837	1.437	1.324	189
wilk20	653	637	1.143	1.006	143
hermite20	1797	1733	3.069	2.668	489
chrma22	5349	5269	10.095	9.590	1273
chrmc23	7189	7061	14.461	13.949	1729

Table 4.3: Variants of the Interval Newton operator. Observe that Variant 2 (v2) is consistently faster and has a marginally smaller subdivision tree in all our tests. The last column contains the number of iterations for which extended interval arithmetic was necessary.

We observe in Table 4.3 that there is very little difference between the two approaches. This is because, in most cases when the Jacobian is not invertible, the results of extended division tend to be an interval with infinite end points ($[-\infty, +\infty]$)[2]. This does not give us any useful information about the presence or absence of roots in a given box. Further, the run time of Variant 1 (with extended division) is always higher because of the extra code required to deal with extended interval arithmetic. Overall, there seems to be *no significant difference between the two approaches, but Variant 2 is faster and simpler to implement.*

Finally, observe that the number of divides by zero (the last column in Table 4.3) is roughly $\frac{1}{4}$ the total number of iterations. This seems to indicate that most subdivisions in Variant 2 are due to the Jacobian being non invertible. This seems to be due to the overestimation of the Jacobian intervals due to the interval extensions, as in the case of the exclusion predicates.

[2]In extended interval arithmetic, $a \div b = [-\infty, +\infty]$ when $0 \in a$ and $0 \in b$.

Cost of recentring: Table 4.3 seems to show a serious degradation in running time as the degree of the polynomial increases. From 0.027s for random degree 4 polynomials, to 10 seconds or higher for degree 20 polynomials. This has been observed to increase to the order of minutes as the degree increases (see Table 4.6). This is largely due to the increased computational cost of calculating the centred forms. In fact, CPU profiling[3] shows:

```
Total: 1291 samples
     956  74.1%  74.1%      978  75.8% std::vector::_M_insert_aux
     268  20.8%  94.8%     1276  98.8% CORE::BiPoly::operator*=
```

The third column is of particular importance to us, it gives the total percentage of time spent in a particular function, and in functions that appear before it. A significant portion of time (nearly 95%) is therefore spent in `Core::BiPoly::operator*=` which is used in the polynomial composition code that we use in our centring. Note that as the degree of the polynomial increases, so does the cost of recentring it.

However, if we recentre less frequently we find that a large number of boxes end up unresolved due to the interval extension overestimating the range of the polynomial over a box. This is best demonstrated visually; see Figure 4.1. Here, areas of blue are unresolved regions, and areas of red are excluded by C_{out}.

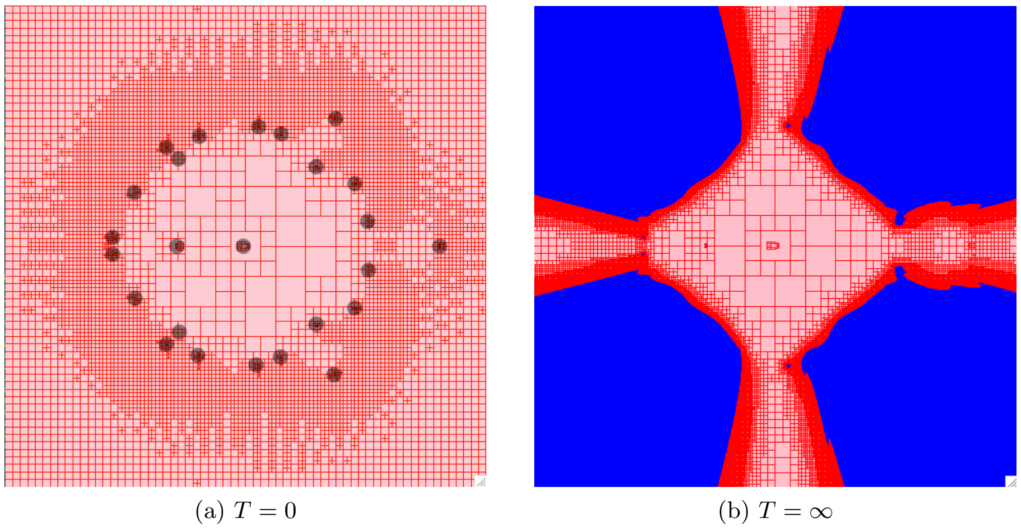

(a) $T = 0$ (b) $T = \infty$

Figure 4.1: Subdivision tree for a degree $n = 25$ polynomial using the Newton operator. The figure is the square $[-2, 2] \times [-2, 2]$. Blue bordered boxes are unresolved.

In Figure 4.1b, we observe that about $\frac{1}{4}$ of the total starting region B_0 is unresolved. This is clearly unacceptable, and if this predicate is to be used on higher degree polynomials, centring seems to be a necessity. Even with centring however, *the performance of the algorithm is unacceptably slow.* We will see in Section 4.3 that the complex analytic predicates can perform much better in these situations.

4.2.3 The Krawczyk operator

In Section 3.4.3 we discussed three variants of the Krawczyk operator. These variants differ in their choice of Y (Eq. (2.21)). Recall that:

- Variant 1 chooses $Y = J^{-1}(\mathbf{y})$.

[3]The profiles were generated using Google Performance Tools [14].

- Variant 2 chooses $Y = \mathrm{mid}(\square J(\mathbf{X})^{-1})$
- Variant 3 chooses $Y = I_{2\times 2}$

Based on our discussion in Sections 2.2.7 and 3.4.3, we expect Variants 1 and 2 to outperform Variant 3 which arbitrarily chooses a constant Y. Recall that in order to produce tighter output intervals, we need to minimise $w(K(\mathbf{X})) = w(\{I - Y \cdot \square J(\mathbf{X})\})$. We would expect that $Y \cdot \square J(\mathbf{X}) \approx I$ as Y gets closer to J^{-1}. As the width of $\mathbf{X}$ reduces, we'd expect Variants 1 and 2 to perform similarly.

	Iters		Time(s)		
	v1/v2	v3	v1	v2	v3
random4	$\underline{332}$	436	$\underline{0.028}$	$\underline{0.028}$	0.037
random6	$\underline{760}$	928	$\underline{0.126}$	0.128	0.160
random8	$\underline{1342}$	1546	0.380	$\underline{0.375}$	0.426
random10	$\underline{1916}$	2148	$\underline{0.841}$	0.842	0.911
nroots10	$\underline{2181}$	2453	$\underline{0.876}$	0.926	0.980
random12	$\underline{2822}$	3070	1.752	$\underline{1.716}$	1.850
chebyshev20	$\underline{13573}$	13675	$\underline{22.087}$	23.725	26.623
nroots20	$\underline{8213}$	8565	$\underline{12.805}$	13.492	15.473
laguerre20	$\underline{829}$	949	$\underline{1.344}$	1.483	1.563
hermite20	$\underline{1733}$	1941	$\underline{2.731}$	2.840	3.066
wilk20	$\underline{629}$	693	$\underline{1.002}$	1.065	1.133
chrma22	$\underline{5245}$	5413	$\underline{9.633}$	9.819	11.991
chrmc23	$\underline{7029}$	7101	$\underline{14.050}$	14.872	14.945

Table 4.4: Variants of the Krawczyk operator. Note that Variants 1 and 2 have very similar performance, though Variant 1 is faster since it costs less to calculate Y in this case.

In Table 4.4 we provide a comparison of the three variants. These tests were run under the same conditions as for the Newton variants in Section 4.2.2. It turns out that in our tests, Variants 1 and 2 are so similar that the number of iterations and splits is exactly the same in both cases. Variant 2 tends to be the slower of the two as it requires the evaluation of $\square J(\mathbf{X})$ and its (interval valued) inverse in order to calculate Y.

Variant 3 is the slowest as expected, due to the selection of a fixed Y. *Among the three Variants, Variant 1 provides the fastest execution and accuracy comparable to Variant 2, and this makes it the best choice.*

4.2.4 Comparison of the Newton type operators

In Sections 4.2.3 and 4.2.2, we compared variants of the Krawczyk and Newton operator respectively. At the end of those sections, we had a clear indication of which variant of each of those operators performed best. We now provide a comparison between the operators of the previous two sections and the Hansen-Sengupta operator. For the Krawczyk and interval Newton operator, the best variant is assumed to have been used (Variant 2 in both cases). Recall that for both of these operators, we do not use extended interval arithmetic and in this respect, they differ from the Hansen-Sengupta operator. We might expect the latter to be slower as a result of this difference. Recall from Section 1.2.1 that we have the following theoretical order on the tightness of output intervals:

$$\text{Interval Newton} > \text{Hansen-Sengupta} > \text{Krawczyk}.$$

The interval Newton operator is therefore expected to perform the best. Table 4.5 contains a comparison of their practical performance.

	Iters			Time(s)			Unresolved		
	N	HS	K	N	HS	K	N	HS	K
random4	331	359	332	0.029	0.032	0.028	36	36	38
random6	767	871	760	0.131	0.147	0.127	40	39	38
random8	1336	1553	1342	0.413	0.439	0.378	40	39	48
random10	1936	2249	1916	0.858	0.971	0.832	39	40	52
nroots10	2213	2485	2181	0.899	0.996	0.866	32	64	88
random12	2819	3295	2822	1.724	1.988	1.706	50	46	55
chebyshev20	13573	16437	13573	21.899	26.231	23.044	160	160	160
nroots20	8245	10837	8213	13.035	16.708	12.753	80	104	80
laguerre20	837	973	829	1.343	1.600	1.322	32	32	32
hermite20	1733	1829	1733	2.713	2.930	2.667	64	64	64
wilk20	637	813	629	1.187	1.330	0.992	24	24	24
chrma22	5269	6405	5245	9.696	12.136	10.067	32	48	32
chrmc23	7061	7973	7029	13.997	16.299	13.745	216	168	352

Table 4.5: Comparison of the three Newton type operators on $[-2, 2] \times [-2, 2]$.

A quick look at Table 4.5 reveals not much difference between the three operators. Their run times and the number of boxes they process are very similar. The Krawczyk operator however, often runs the fastest of the three. The Hansen Sengupta operator is slowed down by its use of extended interval arithmetic, and in most tests is the slowest of the operators that we test.

More disturbing is the fact that each of these algorithms yield unresolved boxes. Even by this metric, there is no clear difference between the three operators. The Newton operator performs slightly better here, and the Krawczyk operator is the worst.

Though the Krawczyk operator is the weakest theoretically (and yields a slightly larger number of unresolved boxes), our tests show that over a wide range of polynomials, it *can be faster* at isolating roots due to it being computationally efficient. Moreover, unlike the Newton operator that is forced to subdivide a box if the interval Jacobian contains a singular matrix, the Krawczyk operator suffers no such "artificial" subdivisions. The operator can always be applied for a suitable choice of Y.

We therefore believe that though overlooked in favour of the Newton or Hansen-Sengupta operator, the Krawczyk operator is a viable interval arithmetic based inclusion predicate.

4.2.5 Higher degree polynomials and unresolved boxes

Previous sections of this chapter provided timing information for polynomials of degree upto 20. We now provide brief results of the application of these operators to higher degree polynomials. Further, we provide some results about the effect of the centring threshold T on the number of unresolved boxes and the overall running time for a higher degree polynomial. In our tests, we use the Krawczyk operator as it performs best as per Section 4.2.4.

From Table 4.6, we see that the performance continues to decrease with increasing polynomial degree. Observe that the number of unresolved boxes increases proportional to the degree as well. We observe that these boxes are clustered around roots (see Figure 4.2). Since the number of boxes is still manageable, we can attempt to resolve them as per Section 3.4.4.

Degree	Iters	Unresolved	Isolated	Time(s)
20	8317	40	16	8.84
30	15065	64	20	31.43
40	31601	152	22	141.74
50	46417	252	20	330.46
60	68509	276	27	750.10

Table 4.6: Performance of the Krawczyk operator with $T = 0$ on higher degree random polynomials

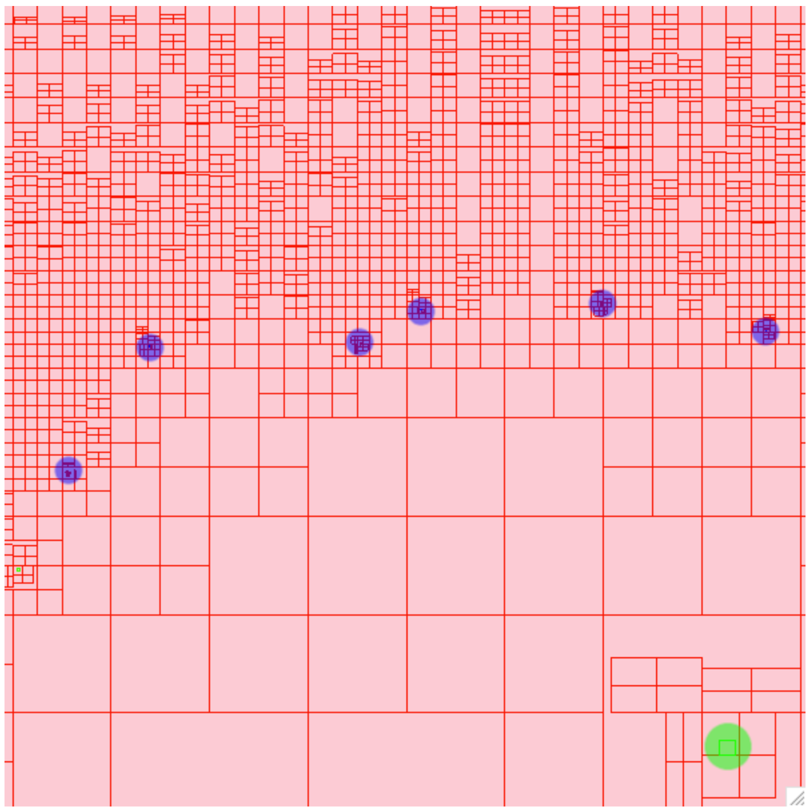


Figure 4.2: Section of the subdivision tree for a degree 40 random polynomial with $T = 0$ showing small clusters of (blue) unresolved regions around roots. The region highlighted with green is a successfully isolated root.

Recall our CPU profiling results from Section 4.2.2 that showed that most of our CPU time was spent centring the polynomial representation. We can try to decrease our running time by centring less frequently, but we would expect our predicate efficiency to decrease. The results of our experiments with a degree 40 random polynomial are the subject of Table 4.7.

T	Iters	Unresolved	Isolated	Time(s)
0	31601	152	22	141.74
4	52681	292	23	18.459
16	94469	3984	11	14.956
64	188741	25708	8	21.668
256	428189	148832	5	35.647
1024	966721	515740	3	60.690

Table 4.7: Performance of the Krawczyk operator on a degree 40 random polynomial with varying values of T.

As expected, the efficiency of the predicate in terms of running time increases, but so does the number of unresolved boxes. In fact, the number of unresolved boxes is so high that a second stage of processing to resolve them would end up processing most of the original area B_0.

This seems to be a major drawback of these interval arithmetic based approaches. *Their predicates appear to be less effective on higher degree polynomials.* This is primarily due to range overestimation by the exclusion predicate. Further, schemes to mitigate some of these problems tend to be computationally expensive. *Lastly, these predicates suffer from issues with roots that lie at box boundaries, for which additional processing is often involved.*

4.3 The CEVAL algorithm

In this section, we discuss the performance of the CEVAL algorithm and the closely related exclusion based approach of Yakoubsohn. We will show that in all situations, these approaches perform better than the approaches of the previous section.

Recall from Section 2.3 that Yakoubsohn's approach does not guarantee isolating boxes and simply outputs non excluded boxes with width $w \leq \varepsilon$. The CEVAL algorithm on the other hand is based on the 8-point test which can guarantee that a box is isolating. Both these approaches use the same exclusion predicate, so any difference in their performance will be due to the application of the 8-point-test and its related preconditions (T'_6, $T'_{\sqrt{2}}$).

We start with a discussion of the exclusion predicate.

4.3.1 The exclusion predicate

Recall the form of our predicate T^f_K from (2.28). Although it involves the evaluation of n partial derivatives for a degree n polynomial, it does not involve any interval operations. Due to this, it does not appear to suffer from the same issues with overestimation as the interval analysis based exclusion predicate. The results are presented in Table 4.8, which contains timings for random polynomials of the specified degree.

Observe that the predicate performance remains stable as the degree of the polynomial is increased until $n = 90$. We do not observe any degradation in the accuracy of the predicate, but as expected the performance decreases since a larger number of derivatives have to be evaluated, and each of those derivatives are themselves higher degree polynomials. On comparing this

Degree	Iters	C_{out} (as % of Iters)	Time(ms)
10	1917	1420(74)	1.699
20	6853	5100(74)	14.610
30	16005	11948(74)	63.833
40	27845	20812(74)	185.072
50	42861	32046(74)	434.557
60	58925	44084(74)	814.191
70	77397	57092(73)	1478.965
80	112445	84168(75)	2790.035
90	142109	106418(75)	4358.963

Table 4.8: Performance of T_K^f with varying polynomial degree. Note that the run time and number of iterations increase, but the predicate is still effective on smaller boxes.

data with Table 4.6 and the discussion in Sections 4.2.5, 4.2.1, we observe significantly better performance that the interval arithmetic based methods. *The times in Table 4.8 are presented in* milli *seconds and are three orders of magnitude faster than the interval methods.*

Issues with degree $n > 90$: This predicate suffers from some pitfalls as the polynomial degree n increases beyond 90. The problem is that the evaluation of this predicate requires us to evaluate each of $1!, 2!, 3!, \dots, n!$, and these factorials grow very fast (see Section 3.5.1).

To work around the problem, we can use approximations of the factorials with correct rounding as described in Section 3.5.1. Even with these approximations, at $n = 90$ our calculations start reaching the thresholds of the machine precision double types. For polynomials higher than this degree, operation at Level 2 is unavoidable. This brings with it the associated slowness of using an extended precision type. See Section 4.3.4 for a comparison of run times at Level 2.

4.3.2 CEVAL performance

We now present the performance of the CEVAL algorithm. Based on our discussion of the exclusion predicate in the previous section, we expect that the CEVAL algorithm will be faster than the interval arithmetic based operators. We present Table 4.9 containing our experimental results. In this case, we attempt to isolate the roots of the test polynomials with the initial box $B_0 = [-2, 2] \times [-2, 2]$.

Naturally, we might also be interested in isolating all roots of the respective polynomials, instead of just those in B_0 defined above. This is the subject of Table 4.10. To isolate all roots, we use the Cauchy bound to estimate a B_0 guaranteed to contain all the roots of the polynomial. This bounding box will be somewhat larger than our fixed choice of $[-2, 2] \times [-2, 2]$ above, so we expect the algorithm to take longer to run as a consequence. However, an effective exclusion predicate (such as T_1) should be able to minimise this difference.

Table 4.10 shows that the run times of CEVAL on an initial box calculated using root bounds are indeed longer than the $B_0 = [-2, 2] \times [-2, 2]$ case. This is expected, since the running time of a subdivision algorithm is dependent on the size of its input box (see the results in Section 4.3.3).

We now turn our attention to the comparison of CEVAL with Yakoubsohn's method. Recall that the 8-point test involves evaluating the polynomial at 8 points on a disk. While these polynomial evaluations are not expensive, the prerequisites for the 8-point test can be. Recall

	Iters	Time(ms)
random10	1909	1.741
nroots10	2037	1.838
random20	7109	14.942
chebyshev20	12805	26.262
nroots20	7989	16.593
laguerre20	805	1.747
hermite20	1685	3.680
wilk20	581	1.371
chrma22	4949	10.978
chrmc23	7101	16.840
random30	16013	62.602
random40	27419	178.732
random50	43160	427.576
random60	60757	843.580
random70	80215	1481.286
random80	111795	2685.381
random90	139605	4211.166

Table 4.9: Performance of the CEVAL algorithm on $B_0 = [-2, 2] \times [-2, 2]$. Note that the run times are in *milli* seconds.

	Iters	Time(ms)
random10	2615	2.342
nroots10	2037	1.816
chebyshev20	18533	39.821
nroots20	7989	16.444
laguerre20	38253	79.055
hermite20	17093	35.618
wilk20	40589	97.393
random20	8201	17.378
chrma22	36749	80.626
chrmc23	43389	107.063
random30	27413	110.893
random40	45722	315.381
random50	71905	743.922
random60	107746	1495.772
random70	108310	2104.210
random80	121129	2946.031
random90	221837	6789.341

Table 4.10: Performance of the CEVAL algorithm in isolating **all** roots. Observe that these are in general higher than Table 4.9.

that $T'_6(m, 4r)$ and $T'_{\sqrt{2}}(m, 8r)$ must hold, and each of these tests are almost the same cost as the exclusion predicate (involving degree $n-1$ polynomials, instead of degree n).

In the case of Yakoubsohn's approach we need to choose a fixed ε, so that we can output boxes whose widths are lesser than ε and are not excluded by T_1. We can choose this value based on a root separation bound (which is a property of the polynomial), or choose a fixed value based on Core Level. We choose the latter for the sake of simplicity, and set $\varepsilon = 0.0001$. Note that as this value becomes smaller, the running time will increase as the number of subdivisions required to reach that size will increase.

The performance comparison between these two approaches can be found in Table 4.11. As expected, *Yakoubsohn's method is faster since it does not bear the burden of calculating* $T'_6(m, 4r), T'_{\sqrt{2}}(m, 8r)$*, and the* 8*-point test. However, in our tests the CEVAL algorithm always operates on a fewer number of boxes.*

We make two observations about Yakoubsohn's method, the first being that it tends to have a larger subdivision tree as it forces boxes to be subdivided until their width is less than a fixed ε. CEVAL can include larger boxes in the output as long as they satisfy the 8-point test. Also, in the case of Yakoubsohn's approach, there are a multiple output boxes clustered around roots. In our tests, polynomials have an average of 8 output boxes around every root. To approximate roots that these boxes are believed to contain, these boxes will need to be processed in some way that the original algorithm does not fully specify.

	Output(Yako.)	Iters		Time(ms)	
		CEVAL	Yako.	CEVAL	Yako.
random10	76	2615	3105	2.721	2.282
nroots10	96	2037	2581	1.773	1.802
chebyshev20	176	18533	19509	38.462	35.544
nroots20	192	7989	8885	16.481	14.526
laguerre20	192	38253	39845	77.991	65.489
hermite20	160	17093	18309	35.690	29.955
wilk20	128	40589	41909	84.321	69.130
random20	147	8201	8985	19.750	14.805
chrma22	168	36749	37661	83.140	67.142
chrmc23	200	43389	43813	101.816	83.464
random30	228	27413	28441	123.294	89.068
random40	296	45722	46957	337.769	245.090
random50	361	71905	73347	783.600	578.127
random60	426	107746	109317	1646.602	1191.557
random70	511	108310	110064	2213.269	1623.679
random80	581	121129	122990	3021.738	2347.435
random90	665	221837	223842	6839.414	5385.514

Table 4.11: Comparison of CEVAL and Yakoubsohn's method with $\varepsilon = 0.0001$. Observe that Yakoubsohn's method is always faster.

Finally, we compare CEVAL with other non subdivision based methods. We choose the MPSolve package, (see Section 2.4.2) which uses the Aberth-Erlich simultaneous iteration. The timings for MPSolve were generated using the UNIX `time` command and are in general not as accurate as the timings generated from our code. In any case, given that these timings are in the order of milli seconds, they are not reliable due to operating system context switches etc.

and are provided as a rough indicator of performance. The list of timings can be found in Table 4.12.

The CEVAL algorithm performs about the same as MPSolve for polynomials with degree $n < 30$. For higher degree polynomials, their performance starts to diverge with MPSolve being consistently faster. At $n = 40$ we see that CEVAL is generally up to five or eight times slower. One of the factors that works against CEVAL is the estimate of B_0 from the Cauchy bound. For instance, the Wilkinson's degree 40 polynomial has an estimated $B_0 = [-2048, 2048] \times [-2048, 2048]$, which is much larger than the minimal bound. The number of iterations of the Aberth-Erlich iteration that are required to converge to a root is not directly related to the root bounds, or to the distribution of roots. The behaviour of this iteration (and the reason for its seemingly wonderful convergence properties) is not very well understood in general; a discussion can be found in [4].

	Time(ms)	
	CEVAL	MPSolve
chebyshev20	39.821	27
laguerre20	79.055	87
hermite20	35.618	81
wilk20	97.393	49
chrma22	80.626	47
chrmc23	107.063	61
hermite40	525.80	88
wilk40	1142.82	153

Table 4.12: Comparison of CEVAL and MPSolve.

We must keep in mind that the Aberth-Erlich (or indeed the Weierstrass-Durand-Kerner) iteration does not output a list of isolating boxes, rather just approximations of roots. We have no guarantee that the iteration will converge to "reasonable" approximations. Our subdivision based methods on the other hand, can produce boxes that are guaranteed to isolate roots. Additionally, the strength of subdivision based methods is that they can operate on tight areas of interest while the simultaneous iteration based methods necessarily have to approximate all roots. Unfortunately, we cannot provide a comparison of the two methods operating on such a preselected area because the MPSolve software does not support such an option (see [5]).

4.3.3 The effect of B_0 on CEVAL performance

In this section, we analyse the performance of the CEVAL algorithm as the size of the input box B_0 increases. We mentioned in earlier sections that we expect the running time to increase with increasing initial box size.

Our test involves the Wilkinson's degree 20 polynomial. This has an estimated bounding box $B_0 = [-512, 512] \times [-512, 512]$ as per the Cauchy bound. We tabulate the running time at Level 1 on boxes of increasing size. Our boxes are of the form $B_0 = [-a, a] \times [-1, 1]$ where $a = \{1, 4, 16, 64, 256\}$. Note that for $a = \{1, 4, 16\}$, some roots of the polynomial will not be isolated as they lie outside the bounding box, and each increase of a increases the area of the initial box by a factor of 4.

The results of this test are presented in Table 4.13. Note that the time increases quite significantly with increasing B_0. This tells us that the speed of our algorithm can potentially

a	Iters	Time(ms)
1	189	1.015
4	2629	12.937
16	27781	134.605
64	172997	834.539
256	351425	1704.078

Table 4.13: The effect of increasing B_0 on algorithm run time.

be improved by choosing a better root bound. This is an area of improvement that we intend to pursue in the longer term, but have insufficient time to include in this thesis.

4.3.4 Running times at Level 2

We now provide a comparison of running times at Level 2 of the operators we discussed. We start with the Newton type operators, whose performance is presented in Table 4.14.

As in the case of Level 1, there is no significant difference between the three operators. At Level 2 however, the interval Newton operator appears to be faster than the Krawczyk operator by a consistently small margin. Given that arithmetic operations are more expensive at this Level, the approach that processes the fewest boxes is bound to be faster. Also, note that the Hansen-Sengupta operator lags further behind the other two as a result of its expensive extended interval arithmetic computations.

Degree	Iters			Time(s)		
	HS	N	K	HS	N	K
4	$\underline{325}$	333	357	$\underline{0.320}$	0.334	0.463
6	877	$\underline{805}$	829	2.285	$\underline{1.769}$	2.164
8	1557	$\underline{1317}$	1349	6.972	$\underline{5.633}$	6.559
10	2461	2093	$\underline{2069}$	19.066	$\underline{15.730}$	17.410
12	3445	$\underline{2965}$	$\underline{2965}$	41.488	$\underline{35.037}$	39.390
14	4677	$\underline{3997}$	4021	87.262	$\underline{71.523}$	77.699

Table 4.14: Comparison of Newton type operators at Level 2 on $[-2, 2] \times [-2, 2]$

As in the case of Level 1, CEVAL continues to be three orders of magnitude faster than the Newton type operators. The results make up Table 4.15.

Degree	Output(Yako.)	Iters		Time(s)	
		CEVAL	Yako.	CEVAL	Yako.
10	72	$\underline{1965}$	2381	$\underline{0.801}$	0.876
20	128	$\underline{7909}$	8565	12.587	$\underline{12.257}$
30	224	$\underline{16413}$	17381	65.162	$\underline{53.582}$
40	248	$\underline{29293}$	30405	203.899	$\underline{163.549}$

Table 4.15: Comparison CEVAL with Yakoubsohn's pure exclusion approach over $[-2, 2] \times [-2, 2]$

The results of this section show that the performance of our algorithms at Level 2 is between 20 and 50 times slower than at Level 1. This is an expected consequence of using extended precision types. To improve performance, it might be advantageous to carry out as many operations as possible at machine precision, and to control precision growth in areas that require it.

The current CEVAL implementation is not capable of switching Core levels at run time for portions of the working set, and can run only entirely at Level 1 or Level 2 as decided at compile time. This is one of the aspects of the implementation that we hope to improve, and we discuss this in Section 5.2.

Chapter 5

Conclusions

5.1 Summary

Our experimental results show that the interval arithmetic based approaches seem to suffer from a serious degradation in performance as the degree n of the polynomial increases. Some of this degradation in performance can be attributed to the overestimation of function range by interval extensions, but methods to compensate for it tend to be computationally expensive. Further, these operators suffer from issues due to roots that lie on box boundaries. Overall, this approach appears to suffer from various practical and performance issues, and its use cannot be recommended.

Among the three interval arithmetic based approaches however, we showed that the Krawczyk operator (despite being the weakest theoretically) did not lag very far behind the other operators in terms of performance. It was in fact the fastest of the three at Level 1 and performed marginally slower than the interval Newton operator at Level 2.

In contrast, our experimental results for the CEVAL algorithm appear quite encouraging. It is efficient and robust, and works well on a larger range of polynomials. It can also provide stronger guarantees than Yakoubsohn's exclusion based approach at a comparable speed. Further, both of these approaches perform three orders of magnitude faster than the interval arithmetic based approaches.

However, the performance of the predicate T_K breaks down due to the growth of $n!$ for polynomials of degree $n > 90$. We discuss possible solutions to this problem in Section 5.2 and we plan to incorporate these solutions into our implementation. Even without these changes, we believe that the algorithm is an efficient and viable choice for isolating roots of complex polynomials of degree $n < 90$.

5.2 Future Work

Most of the discussion in this section revolves around CEVAL. We believe that with some further work, we can extend it to work efficiently with polynomials of higher degree as well.

The 8-point test: Recall that the 8-point-test evaluates the input polynomial at ordinal compass points using the `BigRat` type. Since the `BigRat` type is represented by a pair of `BigInt` instances (a, b) for $\frac{a}{b}$, the exponentiation of these values to a high power will be slow and the representation will be expensive in terms of memory usage as well. Also, since the `BigRat` representation must always contain no common factors, many expensive GCD operations are required as well.

One possible change in this area is to use rounded interval arithmetic to estimate the range of f over the interval lower and upper bound of the approximate value of the ordinal points. Clearly, if the interval is either entirely negative or positive, then so is the sign of f at the exact ordinal point.

The T_K predicate: We need to explore approaches that allow T_K to remain performant for higher degree polynomials as well. Some of the directions we can look in include the truncated evaluation of the Taylor series and controlled precision growth (with correct rounding) of our calculations.

At fixed precision: Note that our implementation runs either entirely at Level 1 or entirely at Level 2. It would be desirable to implement a wrapper over a machine precision type that can detect underflows and overflows, and switch calculation over to Level 2 or a higher Core level only in this situation. We are currently working on the outline of such a type, but a lot of work remains to be done to test it and integrate it with our CEVAL implementation.

We would also like to explore a fixed precision BigFloat that behaves closer to a machine type, but with larger number of bits of precision.

Appendix A

Sample code

We now provide some sample code from our implementation. Our entire contribution to CORE is about 6000 lines of code (excluding unit tests) and can be found as a part of the CORE distribution 2.1 at `http://cs.nyu.edu/exact/core_pages/downloads.html`.

Our code can be found in CORE under :

```
\$CORE_ROOT/corelib2/progs/mesh/krawczyk/...
\$CORE_ROOT/corelib2/progs/mesh/cxy/...
\$CORE_ROOT/corelib2/progs/mesh/ceval/...
\$CORE_ROOT/corelib2/progs/mesh/benchmark/...
\$CORE_ROOT/corelib2/inc/CORE/Wrappers.h
\$CORE_ROOT/corelib2/inc/CORE/ComplexT.h
\$CORE_ROOT/corelib2/inc/CORE/IntervalT.h
```

A.1 The subdivision algorithm

```
void Algorithm::Run() {
  // Statistics gathering.
  unsigned int num_iterations = 0;
  unsigned int num_excludes_T1 = 0;
  unsigned int num_excludes_8P = 0;
  unsigned int num_includes = 0;
  unsigned int num_splits = 0;

  while (!queue_.empty()) {
    ++num_iterations;

    const Box *b = queue_.back();
    queue_.pop_back();

    // Keep the mid and the radius handy.
    const Complex mid = b->mid();
    const double rad = b->radius();

    if (Size(b)) {
      if (!use_inclusion_) {
        // If no inclusion predicate is being used, then
        // if the box is too small - it is a part of the
        // output.
        ambiguous_.push_back(b);
      } else {
        // If an inclusion predicate is being used, then
        // if the box is too small - it is ambiguous.
```

```
          if (display_) {
            ambiguous_.push_back(b);
          } else {
            delete b;
          }
        }
        continue;
      }

      // This is the exclusion predicate.
      if (pred_.T(1, mid, rad)) {
        ++num_excludes_T1;
        if (display_) {
          rejects_.push_back(b);
        } else {
          delete b;
        }
      } else if (use_inclusion_ &&
                 pred_.Tdash(6, mid, 4*rad) &&
                 pred_.Tdash(1.5, mid, 8*rad)) {
        if (pred_.PointTest(mid, 4*rad)) {
          // A success ! . The region can be inserted into the output.
          InsertOutput(mid, 4*rad);
          ++num_includes;
          if (display_) {
            output_b_.push_back(b);
          }
        } else {
          // If the 8 point test fails the region can be
          // excluded.
          ++num_excludes_8P;
          if (display_) {
            rejects_.push_back(b);
          } else {
            delete b;
          }
        }
      } else {
        // Neither excluded or included, therefore split.
        ++num_splits;
        b->Split(&queue_);
        delete b;
      }
    }
  }
```

Appendix B

Command line arguments

B.1 CEVAL

```
$ ./main_ceval --help
USAGE:

   ./main_ceval  [-z <unsigned int>] [-f <unsigned int>] [-g <unsigned
                 int>] [-a] [-M <string>] [-m <string>] [-i] [-d] [-r] [-u
                 <string>] [-y <string>] [-c <string>] [-x <string>] [-p
                 <string>] [--] [--version] [-h]


Where:

   -z <unsigned int>,  --max_coeff <unsigned int>
     Upper bound on magnitude of coefficient

   -f <unsigned int>,  --seed <unsigned int>
     Seed for generating the random polynomial

   -g <unsigned int>,  --degree <unsigned int>
     Degree of random polynomial

   -a,  --random
     Use a random polynomial

   -M <string>,  --max_box_size <string>
     Maximum box size

   -m <string>,  --min_box_size <string>
     Minimum box size

   -i,  --no_use_inclusion
     Do not use the inclusion predicate (Yakoubsohns approach)

   -d,  --display
     Display subdivision tree

   -r,  --use_root_bounds
     Use Cauchy Bounds

   -u <string>,  --y_max <string>
     Maximum y range
   -y <string>,  --y_min <string>
     Minimum y range

   -c <string>,  --x_max <string>
     Maximum x range

   -x <string>,  --x_min <string>
     Minimum x range

   -p <string>,  --poly <string>
     Input polynomial file name

   --,  --ignore_rest
     Ignores the rest of the labeled arguments following this flag.

   --version
     Displays version information and exits.

   -h,  --help
     Displays usage information and exits.
```

B.2 Newton type operators

```
$ ./main_newt --help
USAGE:

   ./main_newt  [-f <unsigned int>] [-a <unsigned int>] [-b] [-e] [-c
                <int>] [-g <int>] [-t <int>] [-l <string>] [-m <string>]
                [-s] [-k] [-n] [-d] [-r] [-u <string>] [-y <string>] [-z
                <string>] [-x <string>] [-p <string>] [--] [--version]
                [-h]


Where:

   -f <unsigned int>,  --seed <unsigned int>
     Seed for generating the random polynomial

   -a <unsigned int>,  --degree <unsigned int>
     Degree of random poly

   -b,  --random
     Use a random polynomial

   -e,  --step_2
     Disambiguate unresolved boxes

   -c <int>,  --core_level <int>
     CORE LEVEL of operation

   -g <int>,  --max_gen_id <int>
     Threshold for recalculating the centered form

   -t <int>,  --cr_threshold <int>
     Threshold for recalculating the centered form

   -l <string>,  --max_box_size <string>
     Maximum subdivision size

   -m <string>,  --min_box_size <string>
```

```
  Minimum subdivision size

-s,  --hansen
  Use the hansen sengupta operator

-k,  --krawczyk
  Use the krawczyk operator

-n,  --newton
  Use newton iteration

-d,  --display
  Display subdivision tree

-r,  --use_root_bounds
  Use Cauchy Bounds

-u <string>,  --y_max <string>
  Maximum y range

-y <string>,  --y_min <string>
  Minimum y range

-z <string>,  --x_max <string>
  Maximum x range

-x <string>,  --x_min <string>
  Minimum x range

-p <string>,  --poly <string>
  Input polynomial file name

--,  --ignore_rest
  Ignores the rest of the labeled arguments following this flag.

--version
  Displays version information and exits.

-h,  --help
  Displays usage information and exits.
```

Appendix C

Polynomials

C.1 The Mignotte like polynomials

So far, we have dealt with polynomials whose roots can be isolated by operating at Level 1. We now present a case (from the MPSolve test suite [5]) where operation at Level 2 is required. Consider the polynomial defined by:

$$p(z) = z^{20} + (100\mathbf{i}z + 1)^3. \tag{C.1}$$

This polynomial has a cluster of three roots very close to $\frac{1}{100}\mathbf{i}$ while the other 17 roots are well separated. The roots in this cluster are separated from each other by a distance of less than 10^{-16}. We try to isolate the roots of (4.1) using CEVAL at Level 1, and get the following output:

```
$ ./main_ceval --display --random \
               --y_min -4 --y_max 4 \
               --x_min -4 --x_max 4 \
               --min_box_size 0.0000000000000000001
iters=9565,includes=32,splits=2391,ambiguous=0,exc_c0=7126,exc_c1=16,
time=325380
Operated on Bounding box : [-4 + (-4)i],[4 + (4)i]
-------------------------
Number of roots:17
<truncated>
-------------Graphic----------------
```

The output shows that we could isolate only 17 of the 20 roots of this polynomial. More troubling is the fact that the output shows no unresolved regions (the variable `ambiguous` in the output above). We would have expected that the roots that could not be isolated would be in unresolved regions, but due to the small value of `min_box_size` specified above, our calculations could not be represented within the precision available and resulted in us wrongly rejecting boxes that contained roots. Running the code at Level 2 produces the right result.

```
$ ./main_ceval --display --random \
               --y_min -4 --y_max 4 \
               --x_min -4 --x_max 4 \
               --min_box_size 0.0000000000000000001
iters=11997,includes=38,splits=2999,ambiguous=0,exc_c0=8960,exc_c1=0,
time=20678781
Operated on Bounding box : [-4 + (-4)i],[4 + (4)i]
-------------------------
Number of roots:20
<truncated>
```

```
m= [3.98986e-16 + (.01)i], r= 2.08167e-17
m= [-4.05925e-16 + (.01)i], r= 2.08167e-17
m= [-3.46945e-18 + (.01)i], r= 2.08167e-17
-------------Graphic----------------
```

We have omitted all output except the cluster of three roots mentioned above[1]. The subdivision tree is shown below. The three clustered roots are at the centre of the figure and are highlighted in blue, while the well separated roots are highlighted in green.

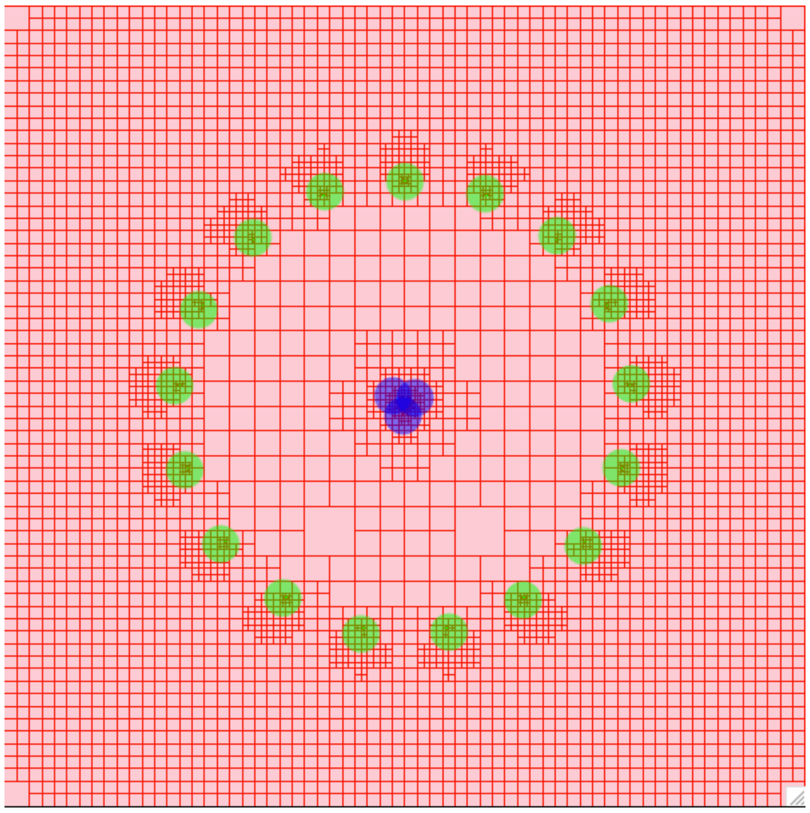


Figure C.1: Subdivision tree for $p(z) = z^{20} + (100\mathbf{i}z + 1)^3$

C.2 CEVAL on non square free polynomials

We mentioned earlier that in order to terminate with all roots isolated, the CEVAL algorithm requires that its input polynomial is square free. We now give an example of CEVAL running on a polynomial that is not square free. In order for the algorithm to terminate, we will need to specify a `min_box_size` and the output will contain unresolved regions around roots that are not simple (of multiplicity 1). For this, we use the polynomial $p(z) = (z^6 + 64)^2(z^6 - 729)$. This polynomial has 6 repeated roots that lie on a disk of radius 2, and 6 simple roots that lie on a disk of radius 3. Running CEVAL on this polynomial, we find that we have 256 unresolved boxes for a `min_box_size` of 0.0001. Further, all 6 simple roots were isolated. The output listing and subdivision tree are given below.

```
iters=8645,includes=16,splits=2161,ambiguous=256,exc_c0=6212,
```

[1]The keen observer might note that this output is missing a text dump of the polynomial like some of the other outputs. This is because the Core library templated polynomial package does not support the display of polynomials with complex coefficients. The changes to do so are currently in the pipeline.

```
exc_c1=0,time=10413791
Operated on Bounding box : [-4 + (-4)i],[4 + (4)i]
With polynomial :
#  -2985984-89216x^{6}-601x^{12}
#   + x^{18}
--------------------------
Number of roots:6
m= [1.498046875 + (-2.599609375)i], r= .01171875
m= [-1.501953125 + (-2.599609375)i], r= .01171875
m= [2.998046875 + (.001953125)i], r= .01171875
m= [1.498046875 + (2.599609375)i], r= .01171875
m= [-3.001953125 + (.001953125)i], r= .01171875
m= [-1.501953125 + (2.599609375)i], r= .01171875
```

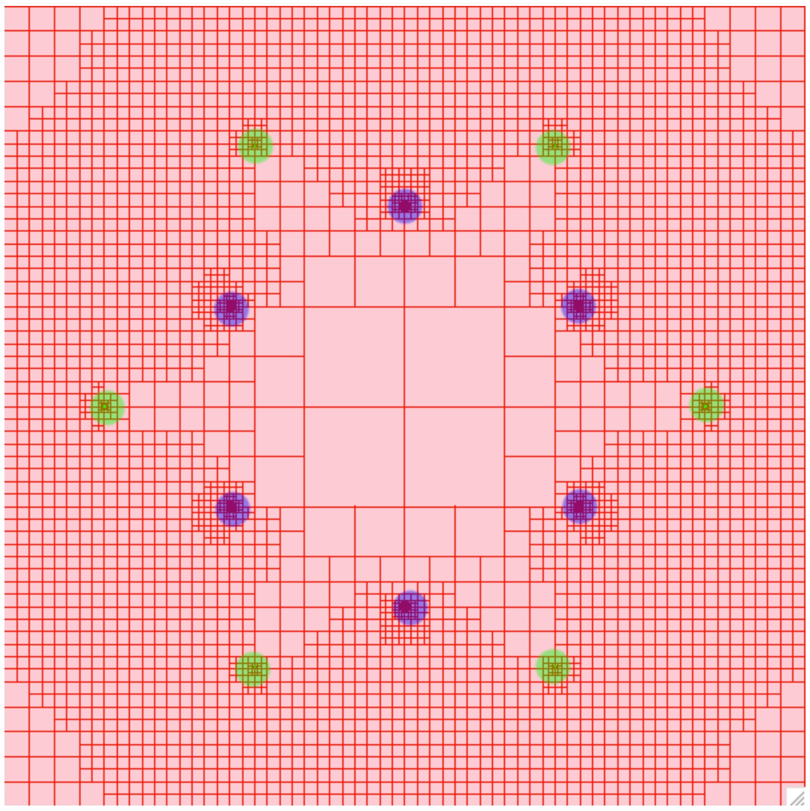


Figure C.2: Subdivision tree for $p(z) = (z^6 + 64)^2(z^6 - 729)$. Areas of blue are unresolved boxes around double roots, and areas of green are isolated simple roots.

C.3 Chebyshev's degree 20 polynomial (chebyshev20)

$$\begin{aligned} p(x) &= 1 - 200x^2 + 6600x^4 - 84480x^6 + 549120x^8 - 2050048x^{10} \\ &+ 4659200x^{12} - 6553600x^{14} + 5570560x^{16} - 2621440x^{18} \\ &+ 524288x^{20} \end{aligned}$$

```
Number of roots:20
m= [0.996948 + (0.00012207)i], r= 0.000732422
m= [0.972168 + (0.000488281)i], r= 0.00292969
m= [0.922852 + (0.000976562)i], r= 0.00585938
```

```
m= [0.853516 + (0.00195312)i], r= 0.0117188
m= [0.759766 + (0.00195312)i], r= 0.0117188
m= [0.650391 + (0.00195312)i], r= 0.0117188
m= [0.521484 + (0.00195312)i], r= 0.0117188
m= [0.380859 + (0.00195312)i], r= 0.0117188
m= [0.232422 + (0.00195312)i], r= 0.0117188
m= [0.0820312 + (0.00390625)i], r= 0.0234375
m= [-0.0820312 + (0.00390625)i], r= 0.0234375
m= [-0.232422 + (0.00195312)i], r= 0.0117188
m= [-0.384766 + (0.00195312)i], r= 0.0117188
m= [-0.521484 + (0.00195312)i], r= 0.0117188
m= [-0.650391 + (0.00195312)i], r= 0.0117188
m= [-0.759766 + (0.00195312)i], r= 0.0117188
m= [-0.853516 + (0.00195312)i], r= 0.0117188
m= [-0.924805 + (0.000976562)i], r= 0.00585938
m= [-0.972168 + (0.000488281)i], r= 0.00292969
m= [-0.996948 + (0.00012207)i], r= 0.000732422
```

C.4 Laguerre's 20th order polynomial (laguerre20)

$$\begin{aligned}
p(x) &= 2432902008176640000 - 48658040163532800000x + 231125690776780800000x^2 \\
&- 462251381553561600000x^3 + 491142092900659200000x^4 \\
&- 314330939456421888000x^5 + 130971224773509120000x^6 \\
&- 37420349935288320000x^7 + 7601008580605440000x^8 \\
&- 1126075345274880000x^9 + 123868287980236800x^{10} - 10237048593408000x^{11} \\
&+ 639815537088000x^{12} - 30287126016000x^{13} + 1081683072000x^{14} \\
&- 28844881920x^{15} + 563376600x^{16} - 7797600x^{17} + 72200x^{18} \\
&- 400x^{19} + x^{20}
\end{aligned}$$

```
Number of roots:20
m= [66.5156 + (0.015625)i], r= 0.09375
m= [55.7969 + (0.015625)i], r= 0.09375
m= [47.6094 + (0.015625)i], r= 0.09375
m= [40.8281 + (0.015625)i], r= 0.09375
m= [35.0156 + (0.015625)i], r= 0.09375
m= [29.9219 + (0.015625)i], r= 0.09375
m= [25.4531 + (0.015625)i], r= 0.09375
m= [21.4844 + (0.015625)i], r= 0.09375
m= [17.9531 + (0.015625)i], r= 0.09375
m= [14.7969 + (0.015625)i], r= 0.09375
m= [12.0469 + (0.015625)i], r= 0.09375
m= [9.57812 + (0.015625)i], r= 0.09375
m= [7.45312 + (0.015625)i], r= 0.09375
m= [5.60938 + (0.015625)i], r= 0.09375
m= [4.04688 + (0.015625)i], r= 0.09375
m= [2.73438 + (0.015625)i], r= 0.09375
m= [1.70312 + (0.015625)i], r= 0.09375
m= [0.921875 + (0.015625)i], r= 0.09375
m= [0.367188 + (0.0078125)i], r= 0.046875
m= [0.0683594 + (0.00195312)i], r= 0.0117188
```

C.5 Hermite polynomial of degree 20 (hermite20)

$$\begin{aligned}p_{20}(x) &= 670442572800 - 13408851456000x^2 + 40226554368000x^4 \\ &- 42908324659200x^6 + 21454162329600x^8 - 5721109954560x^{10} \\ &+ 866834841600x^{12} - 76205260800x^{14} + 3810263040x^{16} \\ &- 99614720x^{18} + 1048576x^{20}\end{aligned}$$

```
Number of roots:20
m= [5.38867 + (0.00195312)i], r= 0.0117188
m= [4.60352 + (0.00195312)i], r= 0.0117188
m= [3.94336 + (0.00195312)i], r= 0.0117188
m= [3.34766 + (0.00390625)i], r= 0.0234375
m= [2.78516 + (0.00390625)i], r= 0.0234375
m= [2.25391 + (0.00390625)i], r= 0.0234375
m= [1.73828 + (0.00390625)i], r= 0.0234375
m= [1.24219 + (0.0078125)i], r= 0.046875
m= [0.742188 + (0.0078125)i], r= 0.046875
m= [0.242188 + (0.0078125)i], r= 0.046875
m= [-0.242188 + (0.0078125)i], r= 0.046875
m= [-0.742188 + (0.0078125)i], r= 0.046875
m= [-1.24219 + (0.0078125)i], r= 0.046875
m= [-1.73828 + (0.00390625)i], r= 0.0234375
m= [-2.25391 + (0.00390625)i], r= 0.0234375
m= [-2.79297 + (0.00390625)i], r= 0.0234375
m= [-3.34766 + (0.00390625)i], r= 0.0234375
m= [-3.94336 + (0.00195312)i], r= 0.0117188
m= [-4.60352 + (0.00195312)i], r= 0.0117188
m= [-5.38867 + (0.00195312)i], r= 0.0117188
```

C.6 Chromatic polynomials (chrma22, chmrc23)

$$\begin{aligned}p_{chrma22}(x) &= -246093 + 2492430x - 12114723x^2 + 37736580x^3 - 84823783x^4 \\ &+ 146716642x^5 - 202994933x^6 + 230138376x^7 - 216927522x^8 \\ &+ 171394508x^9 - 113940788x^{10} + 63782456x^{11} - 30020386x^{12} \\ &+ 11837548x^{13} - 3886852x^{14} + 1052700x^{15} - 231773x^{16} \\ &+ 40562x^{17} - 5445x^{18} + 528x^{19} - 33x^{20} + x^{21}\end{aligned}$$

```
Number of roots:21
m= [1.21289 + (-2.92383)i], r= 0.0117188
m= [2.19434 + (-1.93262)i], r= 0.00585938
m= [2.3291 + (-1.22949)i], r= 0.00585938
m= [2.61914 + (-0.810547)i], r= 0.0117188
m= [2.9873 + (-0.383789)i], r= 0.00585938
m= [0.118164 + (-1.1084)i], r= 0.00585938
m= [1.06934 + (-0.731445)i], r= 0.00585938
m= [1.22168 + (-0.467773)i], r= 0.00585938
m= [1.16602 + (-0.287109)i], r= 0.0117188
m= [1.0752 + (-0.0986328)i], r= 0.00585938
m= [2.9873 + (0.383789)i], r= 0.00585938
m= [2.61914 + (0.810547)i], r= 0.0117188
m= [2.3291 + (1.23145)i], r= 0.00585938
m= [2.19434 + (1.93457)i], r= 0.00585938
```

```
m= [1.0752 + (0.100586)i], r= 0.00585938
m= [1.16602 + (0.291016)i], r= 0.0117188
m= [1.22168 + (0.467773)i], r= 0.00585938
m= [1.06934 + (0.731445)i], r= 0.00585938
m= [0.999512 + (0.000488281)i], r= 0.00292969
m= [0.118164 + (1.11035)i], r= 0.00585938
m= [1.21289 + (2.92773)i], r= 0.0117188
```

$$\begin{aligned} p_{chrmc23}(x) &= 468864 - 4827264x + 23676736x^2 - 73773952x^3 \\ &+ 164369080x^4 - 279584216x^5 + 378695856x^6 - 420778328x^7 \\ &+ 392030336x^8 - 311242152x^9 + 212910190x^{10} - 126266334x^{11} \\ &+ 65006540x^{12} - 28961860x^{13} + 11085034x^{14} - 3604758x^{15} \\ &+ 981064x^{16} - 219016x^{17} + 39018x^{18} - 5330x^{19} + 524x^{20} \\ &- 33x^{21} + x^{22} \end{aligned}$$

C.7 Wilkinson's polynomials (wilk20)

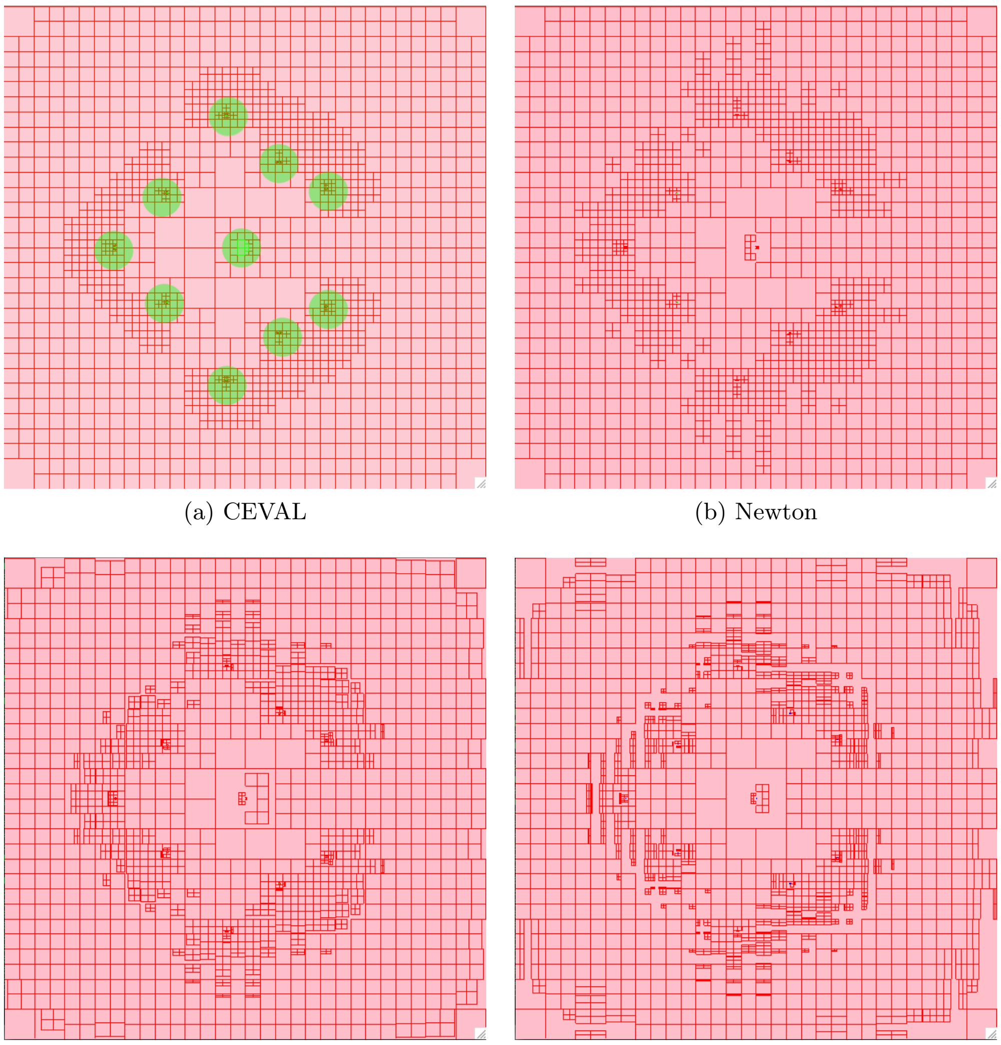
$$\begin{aligned} p(x) &= 2432902008176640000 - 8752948036761600000x + 13803759753640704000x^2 \\ &- 12870931245150988800x^3 + 8037811822645051776x^4 - 3599979517947607200x^5 \\ &+ 1206647803780373360x^6 - 311333643161390640x^7 + 63030812099294896x^8 \\ &- 10142299865511450x^9 + 1307535010540395x^{10} - 135585182899530x^{11} \\ &+ 11310276995381x^{12} - 756111184500x^{13} + 40171771630x^{14} - 1672280820x^{15} \\ &+ 53327946x^{16} - 1256850x^{17} + 20615x^{18} - 210x^{19} + x^{20} \end{aligned}$$

```
Number of roots:20
m= [19.998046875 + (.001953125)i], r= .01171875
m= [18.99609375 + (.00390625)i], r= .0234375
m= [17.99609375 + (.00390625)i], r= .0234375
m= [16.9921875 + (.0078125)i], r= .046875
m= [15.9921875 + (.0078125)i], r= .046875
m= [14.9921875 + (.0078125)i], r= .046875
m= [14.015625 + (.015625)i], r= .09375
m= [12.984375 + (.015625)i], r= .09375
m= [11.984375 + (.015625)i], r= .09375
m= [10.984375 + (.015625)i], r= .09375
m= [9.984375 + (.015625)i], r= .09375
m= [8.984375 + (.015625)i], r= .09375
m= [7.984375 + (.015625)i], r= .09375
m= [6.984375 + (.015625)i], r= .09375
m= [5.9921875 + (.0078125)i], r= .046875
m= [4.9921875 + (.0078125)i], r= .046875
m= [3.9921875 + (.0078125)i], r= .046875
m= [2.99609375 + (.00390625)i], r= .0234375
m= [1.99609375 + (.00390625)i], r= .0234375
m= [0.998046875 + (.001953125)i], r= .01171875
```

Appendix D

Subdivision trees

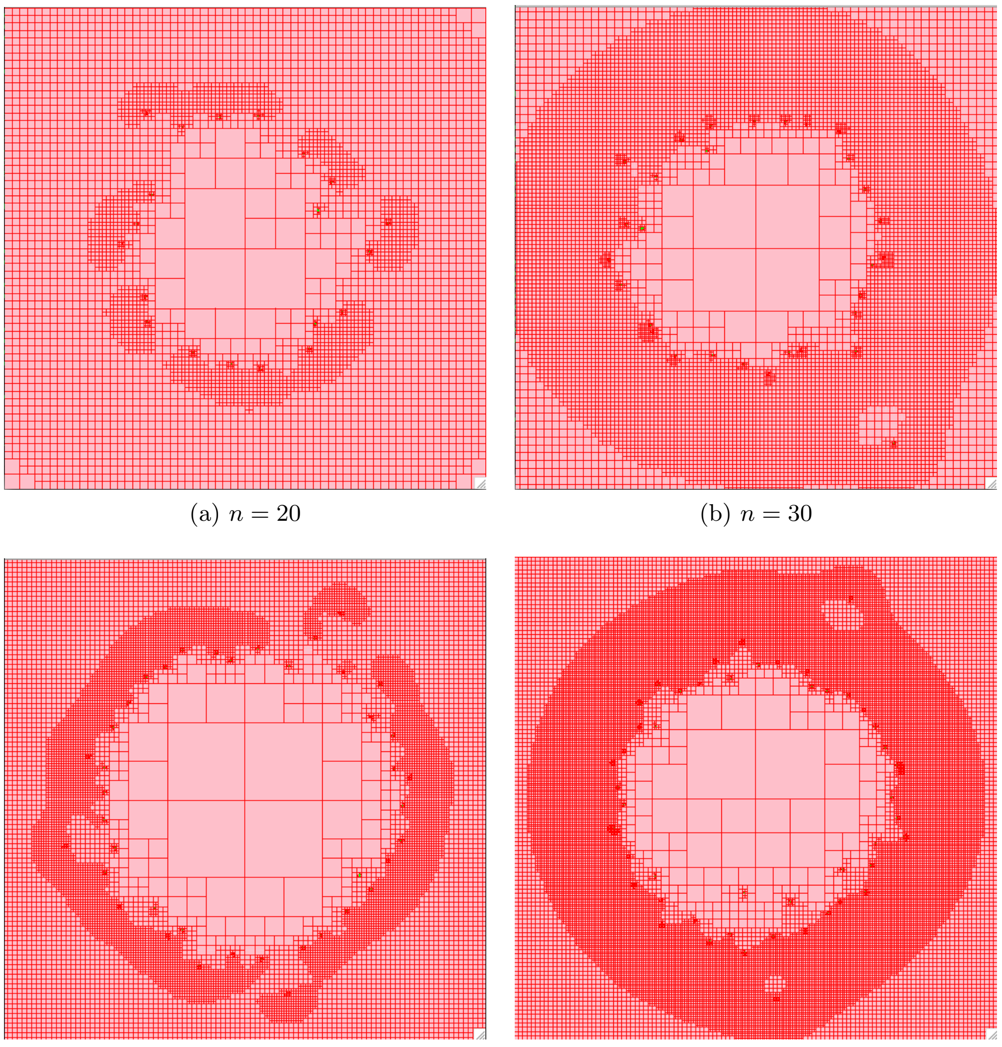
D.1 $x(9x^9 + 7x^8 + 8x^7 + 8x^6 + 6x^4 + 5x^3 + 2x^2 + x) = 0$


(a) CEVAL (b) Newton

(c) Krawczyk (d) Hansen-sengupta

Figure D.1: Subdivision trees for a degree 10 polynomial.

D.2 CEVAL with increasing degree



(a) $n = 20$

(b) $n = 30$

(c) $n = 40$

(d) $n = 50$

Figure D.2: Subdivision trees for randomly generated degree n polynomials.

Appendix E

Music & other inconsequential things

Music: This is a summary of the music I have listened to through the writing of this thesis. If there is ever a scientific study about the effect of music on creative or scientific output, then this shall serve as a data point.

1. Eat a peach, The Allman Brothers band (Southern rock, Blues rock)
2. Fresh cream, Cream (Blues rock, psychedelic rock)
3. Baptizm of Fire, Glenn Tipton (Heavy metal, Hard rock)
4. Maktub, Motherjane (Progressive rock)
5. Mer de Noms, A perfect circle (Progressive rock, Alternative rock)
6. One hot minute, Red hot chilli peppers (Alternative rock, Funk rock)
7. Born Again, Black Sabbath (Heavy metal)

Tigers: I would also like to draw attention to the plight of tigers worldwide. Their numbers are fast dwindling due to habitat destruction and wanton poaching for "medicinal" reasons, or maybe because someone wants a cool carpet. Something needs to be done before these beautiful and graceful creatures vanish from our planet forever. Dont forget, 2010 is the year of the tiger: `http://www.worldwildlife.org/species/finder/tigers/year-of-tiger.html`.

Bibliography

[1] O. Aberth. Iteration methods for finding all zeros of a polynomial simultaneously. *Mathematics of Computation*, 27:339–344, 1973.

[2] G. Alefeld. *Intervallrechnung iiber den komplexen Zahlen und einige Anwendungen.* Doctoral dissertation, University of Karlsruhe, 1968.

[3] G. Alefeld and J. Herzberger. *Introduction to Interval Computations.* Academic Press, New York, 1983. Translated from German by Jon Rokne.

[4] D. A. Bini. Numerical computation of polynomial zeroes by means of aberth's method. *Numerical Algorithms*, 13:179–200, 1996.

[5] D. A. Bini and G. Fiorentino. MPSolve: manual for the MPSolve package. `http://www.dm.unipi.it/cluster-pages/mpsolve/mpsolve.pdf`.

[6] D. A. Bini and G. Fiorentino. MPSolve: numerical computation of polynomial roots v2.2, 2001.

[7] M. Burr, S. Choi, B. Galehouse, and C. Yap. Complete subdivision algorithms, II: Isotopic meshing of singular algebraic curves. In *Proc. Int'l Symp. Symbolic and Algebraic Computation (ISSAC'08)*, pages 87–94, 2008. Hagenberg, Austria. Jul 20-23, 2008. Accepted for Special Issue of ISSAC 2008 in JSC.

[8] CORE library v2.1, download, August 2010. `http://cs.nyu.edu/exact/core/download/core/`.

[9] Core Library homepage, since 1999. Software download, source, documentation and links: `http://cs.nyu.edu/exact/core/`.

[10] J.-P. Dedieu and J.-C. Yakoubsohn. Localization of an algebraic hypersurface by the exclusion algorithm. *Applicable Algbebra in Engineering, Communication and Computing*, 2:239–256, 1992.

[11] The GNU Multi Precision Library, Since 1991, v5.0.1 : February 2010. `http://gmplib.org`.

[12] GNU MP Online Manual, 2010, Free Software Foundation, Inc. `http://gmplib.org/manual/`.

[13] S. Goedecker. Remark on algorithms to find roots of polynomials. *SIAM J. Sci. Comput.*, 15:1059–1063, 1994.

[14] Google Inc. Google perftools. `http://code.google.com/p/google-perftools`.

[15] E. R. Hansen. On solving systems of equations using interval arithmetic. *Math. Comp.*, 22, 1968.

[16] E. R. Hansen. A globally convergent interval method for computing and bounding real roots. *BIT*, 16:415–424, 1978.

[17] E. R. Hansen and S. Sengupta. Bounding solutions of systems of equations using interval analysis. *BIT*, 21:203–211, 1981.

[18] E. R. Hansen and R. R. Smith. Interval arithmetic in matrix computations, part ii. *SIAM J. Numer. Anal.*, 4:1–9, 1967.

[19] IEEE 754 committee. 754-2008 IEEE Standard for Floating-Point Arithmetic, 2008.

[20] ISO/IEC. ISO/IEC 14882:2003(E):Programming Languages - C++, 2003.

[21] M. Jenkins and J. Traub. A three-stage variable shift iteration for polynomial zeros and its relation to generalized rayleigh iteration. *Numer. Math.*, 14:252–263, 1970.

[22] An interview with William Kahan by Charles Severance, Feb. 1998. `http://www.eecs.berkeley.edu/~wkahan/ieee754status/754story.html`.

[23] I. Kerner. Ein Gesamtschrittverfahren zur Berechnung der Nullstellen von Polynomen. *Numerische Mathematik*, 8:290–294, 1966.

[24] R. Krawczyk. Newton-Algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken. *Computing*, 4:187–201, 1969.

[25] L. Lin and C. Yap. Adaptive isotopic approximation of nonsingular curves: the parametrizability and non-local isotopy approach. In *Proc. 25th ACM Symp. on Comp. Geometry*, pages 351–360, June 2009. Aarhus, Denmark, Jun 8-10, 2009. Accepted for Special Issue of SoCG 2009 in DCG.

[26] J. McNamee. A bibliography on roots of polynomials. *J. Comput. Appl. Math.*, 47:391–394, 1993. Available online at http://www.elsevier.com/homepage/sac/cam/mcnamee.

[27] R. Moore and S. Jones. Safe starting regions for iterative methods. *SIAM J. Num.Analysis*, 14(6):1051–1065, 1977.

[28] R. E. Moore. *Interval Analysis.* Prentice Hall, Englewood Cliffs, NJ, 1966.

[29] R. E. Moore. A test for existence of solution to nonlinear systems. *SIAM J. Numer. Anal.*, 14:611–615, 1977.

[30] R. E. Moore and L. Qi. A successive interval test for nonlinear systems. *SIAM J. Numer. Anal.*, 19:845–850, 1982.

[31] The MPFR Library, v3.0.0 : June 2010. `http://www.mpfr.org`.

[32] The MPFR Library 3.0.0 Manual, June 2010. `http://www.mpfr.org/mpfr-current/mpfr.html`.

[33] J. R. Munkres. *Topology.* Prentice Hall, second edition, 2000.

[34] A. Neumaier. *Interval Methods for Systems of Equations.* Cambridge University Press, Cambridge, 1990.

[35] K. Nickel. On the Newton Method in Interval Analysis. Research Memorandum MRC Technical Summary Report #1136, University of Wisconsin, Madison, 1971.

[36] S. Plantinga and G. Vegter. Isotopic approximation of implicit curves and surfaces. In *Proc. Eurographics Symposium on Geometry Processing*, pages 245–254, New York, 2004. ACM Press.

[37] M. Sagraloff and C. K. Yap. An efficient exact subdivision algorithm for isolating complex roots of a polynomial and its complexity analysis, July 2009. Submitted.

[38] V. Stahl. *Interval Methods for Bounding the Range of Polynomials and Solving Systems of Nonlinear Equations.* Ph.D. thesis, Johannes Kepler University, Linz, 1995.

[39] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis.* Springer, third edition, 2002.

[40] The FRISCO Consortium. FRISCO Polynomial test suite. `http://www-sop.inria.fr/saga/POL/`.

[41] I. Voiculescu. *Implicit function algebra in set theoretic geometric modelling.* PhD Thesis, University of Bath, 2001.

[42] J.-C. Yakoubsohn. Numerical analysis of a bisection-exclusion method to find zeros of univariate analytic functions. *J. of Complexity*, 21:652–690, 2005.

[43] C. K. Yap. *Fundamental Problems of Algorithmic Algebra.* Oxford University Press, 2000.

[44] C. K. Yap. Tutorial: Exact numerical computation in algebra and geometry. In *Proc. 34th Int'l Symp. Symbolic and Algebraic Comp. (ISSAC'09)*, pages 387–388, 2009. KIAS, Seoul, Korea, Jul 28-31, 2009.

[45] V. Y.Pan. Solving a polynomial equation: Some history and recent progress. *SIAM Review*, 39:187–220, 1997.

[46] J. Yu, C. Yap, Z. Du, S. Pion, and H. Bronnimann. Core 2: A library for Exact Numeric Computation in Geometry and Algebra. In *3rd Proc. Int'l Congress on Mathematical Software (ICMS)*, 2010. To appear.