Computing the Zeros of Polynomials using the Divide and Conquer Approach

Michael R Farmer
Department of Computer Science and Information Systems
Birkbeck
University of London

PhD Thesis

24th September 2013
(Revised 1st July 2014)

This is to certify that this PhD thesis is, to the best of my knowledge, entirely my own work, except where explicitly stated in the text, and that it has not been submitted, either in part or whole, for a degree at this or any other University.

Signed .....

Abstract

The thesis describes two algorithms for locating all the zeros of arbitrary real or complex polynomials. This approach consists of two distinct stages.

The first stage is called the search stage. Here, the complex plane is searched systematically for regions containing zeros of the original polynomial.

This is essentially the "Divide and Conquer Approach" of the title. Regions containing zeros are sub-divided into smaller regions which may contain zeros. During this process, those regions that do not contain zeros are discarded.

This process is iteratively refined until the total number of zeros in the regions, i.e. those containing zeros, equals the degree of the polynomial. The centres of these regions are therefore approximations to the zeros themselves, some of which may be multiple in number.

At this point, the algorithm switches to the second, or iterative, stage. Here, Iteration Functions (IFs) are used to accelerate convergence to the values of the zeros (to computational precision).

Whilst carrying out this research we discovered new families of IFs that do not appear in the technical literature. The derivations of these IFs are in the body of the thesis. These IFs are showcased by showing their outputs for each of the polynomials tested. These outputs demonstrate the different orders of covergence of the IFs and other interesting features, explanations of which are included in Chapter 7, starting on page 87, of the thesis.

A database of over 200 poynomials was built up and its contents are listed in Chapter 6, starting on page 66, and the results of applying our two-stage process to these polynomials are summarised in the thesis. Some of these polynomials are of high degree.

Contents

List of Figures

List of Tables

Acknowledgments

Professor George Loizou

I especially thank George for taking me on in the first place. He has always been a strong supporter for us working together. However, after our first batch of papers, I drifted into UNIX and open source projects. Stoically, he said nothing.

Many years later, when a surprise 70th birthday party was organised for him at Royal Holloway College in Egham, Surrey, by his daughter Melanie and one of his ex-students, I was surrounded by his past and present students. I just felt it was time to complete what we had started so many years ago. I asked him outright whether he would be willing to take me on a second time in order to finish what we had started. Without hesitation, he said yes. This is the culmination of the work since then. I really thank George for his encouragement and drive to complete this work.

We have worked together, successfully, for many years and, as I have said, he has been behind me all the way. For this reason I have included his name together with mine in describing the work I have undertaken.

Sue Small

Obviously, before I could start, I had to talk to my wife Sue about the work and time involved. She readily agreed in spite of the disruption to our social life this would cause. Luckily, she has her own interests outside her work, and this meant we could survive my intensive work load.

In spite of this, she was still willing to read an earlier draft of this thesis for grammatical and syntactic errors on my part. Thank you, Bubs.

Professor Steve Maybank

At Birkbeck we implement a system where each research student has two supervisors. Steve agreed to be my second supervisor. He read numerous early drafts of this thesis but, more importantly, introduced me to software

packages that could perform symbolic manipulation, which could be used to generate the equations that we were using.

His enthusiasm for such tools led me to learning Matlab [Mat12], which we now use to generare and validate our IFs and their orders of convergence.

Professor Trevor Fenner

Trevor subtly demonstrating to me that a zero value on a page of printout didn't necessarily mean a zero value in the corresponding memory location.

It is something I should have remembered from my undergraduate days, but sadly had forgotten.

My Dad

My father always believing that I could achieve something worthwhile after graduating with a BSc followed by an MSc. He saw this PhD thesis as the next step. I was the first person in our family to go to University and he was immensely proud of what I had achieved so far. Unfortunately, he died before I could satisfy his hope of seeing better things.

And also...

Donald Knuth

For giving us TEX [Knu84].

Leslie Lamport

For giving us LATEX [Lam94], the vehicle for producing this thesis without any hassle.

I trust I have not left out anyone who was also important to me. Obviously, any omissions or factual errors are entirely mine.

Chapter 1

Introduction

The problem of locating the zeros of arbitrary polynomials has a history going back to at least 2000BC when the Babylonians found a general solution for the zeros of quadratic equations, see Agernon Berriman's article [Ber56, pp. 185–186] for details.

1.1 Brief History and Background

The next advance came in 1079 when Omar Khayyám (probably better known for his collection of poems, the *Rubáiyát*) invented a method for solving cubic equations using geometry.

Around 1400 one Al-Kashi could solve specific cubic equations using iteration, and in 1484 Nicholas Chuqet invented a method for solving general polynomials iteratively. Unfortunately, we have no further details.

In 1545 Girolamo Cardano presented the general solution for cubic equations in his Ars Magna [Car45]. He was also the first author to use complex numbers, although these were only used in his derivation of real solutions.

By the 17th century researchers were investigating various properties of polynomials that could be used in understanding the properties of general polynomials.

By the end of the 18th century it was agreed that only general polynomial equations of degree four or lower could be solved using explicit formulas.

In 1831, Augustin-Louis Cauchy determined how many roots of a polynomial lie inside a given contour in the complex plane.

Further references to the above brief history can be found in Victor Pan's survey article [Pan97].

This is the beginning of our work searching the complex plane for regions containing zeros of a given polynomial, see §5.2, starting on page 43.

In the 1970s, when we started work in this area we concentrated on Iteration Functions (IFs) because computers were slow and access was limited. Starting with initial approximations to the zeros meant that the computers we had at the time (expensive main frames) could produce results within a few days, sometimes a week, or so.

By the 1980s it became apparent that careful *cherry picking* of initial approximations could make certain results better than others. Therefore, we decided to include a search stage that identified potential initial approximations by searching the complex plane automatically. This made the choice of initial approximations to our IFs programmable, rather than chosen by researchers. The disadvantage of this approach was that it took a long time and checkpoint/restart facilities were needed through the slow search stage.

By the first decade of this century we had more raw computing power on the PCs under our desks than we could have imagined less than a decade before. This meant that computing time was no longer relevant. We could use multiple-precision libraries, such as [Gra11], without worrying about computing time. Results appear within seconds of inputing the original polynomial, although higher-degree polynomials can take rather longer. In fact, the search stage for our polynomial of degree 400 took 2 hours, 25 minutes, and 4.42 seconds! See Table 7.3 on page 98 for more details about timimgs.

1.2 Motivation and Aims

Our motivation for writing this thesis is to put on record the work we have done over the years developing a general algorithm for locating the zeros of arbitrary real and complex polynomials.

Our aim is to provide both the algebraic background used in deriving the algorithm and to deliver a realisation of this in the form of a set of computer programs that other researchers can use to compare our approach with theirs.

1.3 Structure of the Thesis

Chapter 2, starting on page 5, contains common definitions and describes the notation we have used throughout this thesis.

Chapter 3, starting on page 13, covers the work performed by others in the past and linking their work to ours where it is relevant.

Chapter 4, starting on page 27, describes the work currently being done by researchers in the same field as us. Again, it links their work to ours where appropriate.

Chapter 5, starting on page 42, describes our current research including a global search algorithm, i.e. the search stage, for locating regions containing initial approximations to the zeros of the polynomials under consideration. It then presents a number of IFs (old and new), i.e. the iterative stage, that we have derived which, using the initial approximations mentioned above, converge to the values of the zeros (to computational precision).

Chapter 6, starting on page 66, describes our database of over 200 polynomials that we have collected from numerous sources giving their actual zeros, where known, and references to where they can be found.

Chapter 7, starting on page 87, describes the computational results when our two-stage algorithm is applied to all of the polynomials in the database. It then indicates whether the search stage was successful and, if so, the number of iterations taken by each of the IFs to converge. If convergence is not successful there is a brief description why we think this occurred.

Chapter 8, starting on page 101, gives our thoughts on what we have achieved with the material described in this thesis and proposes some further interesting research topics in this area.

Chapter 9, starting on page 104, decribes the new research we intend to carry out in order to improve our algorithms and a number of different approaches we intend to experiment with in order to make the overall algorithms more robust in terms of their ability to converge. Some may ask what we mean by *robust*, and the best definition we have found so far is that by Michael Jenkins and Joseph Traub in [JT75, p. 28], namely

> "By robustness we mean the ability of a program to degrade gracefully near the boundary of the problem space where the algorithm applies."

Appendix A, starting on page 107, is concerned with deriving the orders of convergence of the different classes of IFs. Supporting equations are also derived.

Appendix B, starting on page 120, contains the *rational* forms of the IFs, expressed in *polynomial* form, contained in Appendix A, starting on page 107.

Appendix C, starting on page 125, contains the listings of all programs used to implement our underlying algorithms.

Finally, starting on page 214, there is a Bibliography of all the resources we have accessed (e.g. books, journals, papers, software packages, etc.) while undertaking our basic research and preparing this thesis.

Chapter 2

Preliminaries

2.1 Notation

Table 2.1 shows the notation used in this thesis.

Construct	Meaning
$(\cdot)^T$	The transpose of $(\cdot)$
$(\cdot)^H$	The Hermetian adjoint (i.e. conjugate transpose) of $(\cdot)$
a_i	The coefficients of $p(z),\ i = 0,\ 1,\ \ldots,\ n$
$\bar{a}_i$	The complex conjugate of a_i
α_i	The zeros of $p(z),\ i = 1,\ 2,\ \ldots,\ N,\ \ N \leq n$
C	Asymptotic error constant
$C(p)$	The **companion matrix** associated with $p(z)$
$C_k(p)$	The kth order **convolution matrix** associated with $p(z)$
$\deg(p)$	The degree of $p(z)$
$\gcd(p,\ q)$	The Greatest Common Divisor of the polynomials $p(z)$ and $q(z)$
$\mathbf{i}$	$\mathbf{i}^2 = -1$
IF	Iteration Function
M	The maximum multiplicity of the zeros of $p(z)$
m_i	The multiplicity of the zero α_i
N	The number of distinct zeros of $p(z)$
n	The degree of $p(z)$
$p(z)$	A polynomial of degree n
$p'(z)$	The first derivative of $p(z)$ of degree $n-1$
$p^{(k)}(z)$	The kth derivative of $p(z)$ of degree $n-k$
$\mathrm{rad}(c)$	The radius of the circle c
$S(p,\ q)$	The **Sylvester resultant matrix** associated with $p(z)$ and $q(z)$
$S_k(p)$	The kth order **Sylvester discriminant matrix** associated with $p(z)$
z_ν	The current approximation to α_ν
$\hat{z}_\nu$	The next approximation to α_ν

Table 2.1: Our Notation

The same letter in boldface, e.g. $\mathbf{p}$, denotes the coefficient (column) vector of $p(z)$, namely

$$\mathbf{p} = (a_n,\ a_{n-1},\ \ldots,\ a_0)^T \quad , \tag{2.1}$$

unless defined otherwise. However, note the use of $\mathbf{i}$ as $\sqrt{-1}$ in Table 2.1 on page 5.

2.2 Definitions

Throughout this thesis sums and products will be over the range $1,\ 2,\ \ldots,\ N$ unless stated otherwise, i.e.

$$\begin{aligned} \sum_{i<\nu} a_i &= \sum_{i=1}^{\nu-1} a_i \quad , \\ \prod_{i\neq\nu} b_i &= \prod_{\substack{i=1 \\ i\neq\nu}}^{N} b_i \quad . \end{aligned} \tag{2.2}$$

Let $p(z)$ be a polynomial of degree n given by

$$p(z) = a_n z^n + a_{n-1} z^{n-1} + \cdots + a_0, \quad a_n a_0 \neq 0 \quad , \tag{2.3}$$

where the coefficients are complex numbers and the zeros $\{\alpha_i\}$ have multiplicities $\{m_i\}$, respectively.

When $p(z)$ is **monic**, $a_n = 1$ and

$$p(z) = \prod_i (z - \alpha_i)^{m_i} \quad . \tag{2.4}$$

Note that N is the number of distinct zeros of $p(z)$. We next define

$$M = \max_i m_i \quad . \tag{2.5}$$

Let $p(z)$ be defined as in Equation (2.3) on page 6. The following definitions, originally given by Joseph Traub [Tra64, pp. 5–6], will be used subsequently.

$$u(z) = \frac{p(z)}{p'(z)} \quad , \tag{2.6}$$

which he calls the *normalised* $p(z)$, and

$$A_i(z) = \frac{p^{(i)}(z)}{i! p'(z)} \quad , i = 1,\ 2,\ \ldots,\ N \quad , \tag{2.7}$$

where $p^{(i)}(z)$ is the ith derivative of $p(z)$ and he calls the *normalised Taylor series cofficient*. Note that $u(z)$ is often referred to as **Newton's correction** [Pet89, p. 85]. For our later use it is worth noting that

$$A_i'(z) = (i+1)A_{i+1}(z) - 2A_2(z)A_i(z) \quad , i = 1,\ 2,\ \ldots \quad , \tag{2.8}$$

which will be used in subsequent derivations.

Let z_ν be an approximation to the zero α_ν, and $\hat{z}_\nu$ be the next approximation to α_ν, using some iterative scheme; we now define

$$\begin{aligned} \epsilon_\nu &= z_\nu - \alpha_\nu, \\ \hat{\epsilon}_\nu &= \hat{z}_\nu - \alpha_\nu, \end{aligned} \quad \nu = 1,\ 2,\ \ldots,\ N \quad , \tag{2.9}$$

with

$$\epsilon = \max_i |\epsilon_i| \quad . \tag{2.10}$$

The following definitions are useful when deriving the order of convergence of our IFs.

$$S_k(z_\nu) = \sum_{i \neq \nu} \frac{m_i}{(z_\nu - \alpha_i)^k} \quad , k = 1,\ 2,\ \ldots \quad . \tag{2.11}$$

Note that this *does not* follow Alston Householder's convention [Hou70, p. 177] of retaining all terms in the summation. This avoids the possible problem when z_ν takes the value α_ν. Again it is worth noting that

$$S_k'(z_\nu) = -kS_{k+1}(z_\nu) \quad , k = 1,\ 2,\ \ldots \quad , \tag{2.12}$$

which will be used in subsequent derivations.

The following definition forms an integral part of our IFs for computing zeros simultaneously.

$$T_k(z_\nu) = \sum_{i \neq \nu} \frac{m_i}{(z_\nu - z_i)^k} \quad , k = 1,\ 2,\ \ldots \quad , \tag{2.13}$$

where, again, this summation avoids a possible term yielding division by zero.

Let $p(z)$ be defined as in Equation (2.3) on page 6. The **reciprocal polynomial** of $p(z)$ [Hen74, p. 492] is defined by

$$p^*(z) = \bar{a}_0 z^n + \bar{a}_1^{n-1} z^{n-1} + \cdots + \bar{a}_n \quad , \tag{2.14}$$

where $\bar{a}_i$ denotes the complex conjugate of a_i. The zeros of $p^*(z)$ are, relative to the unit circle $|z| = 1$, the complex conjugate inverses of the zeros of $p(z)$ [Hen74,

pp. 492–493], i.e. $\frac{1}{\alpha_\nu}$.

Let $p(z)$ be defined as in Equation (2.3) on page 6 and let $p^*(z)$ be defined as in Equation (2.14). Define

$$T(p(z)) = \bar{a}_0 p(z) - a_n p^*(z) \quad , \tag{2.15}$$

which has no term in z^n, so the degree of $T(p(z))$ is lower than that of $p(z)$. This is known as the **Schur transform** of $p(z)$ [Hen74, p. 493]. The constant term of $T(p(z))$ is real, namely

$$T(p(0)) = \bar{a}_0 a_0 - a_n \bar{a}_n = |a_0|^2 - |a_n|^2 \quad . \tag{2.16}$$

The transformation T can be applied to $T(p(z))$ to define the **iterated Schur transforms** given by

$$T(p(z)),\ T^2(p(z)),\ \ldots,\ T^n(p(z)) \quad , \tag{2.17}$$

where $T^i(p(z))$ is regarded as a polynomial of degree $n - i$ even if its leading coefficient is zero [Hen74, p. 493]. We set

$$\gamma_i = T^i(p(0)), \quad i = 1,\ 2,\ \ldots,\ n \quad . \tag{2.18}$$

2.2.1 Companion Matrix

Our *monic* polynomial $p(z)$ has an associated **companion matrix** defined by

$$C(\mathbf{p}) = \begin{bmatrix} 0 & 0 & \ldots & 0 & -a_0 \\ 1 & 0 & \ldots & 0 & -a_1 \\ 0 & 1 & \ldots & 0 & -a_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & -a_{n-1} \end{bmatrix} \quad , \tag{2.19}$$

whose eigenvalues are the zeros of $p(z)$, the **characteristic polynomial** of the square matrix [Mar66, p. 140]. Occasionally this matrix is defined as $C^T(\mathbf{p})$.

The **characteristic equation** is defined by

$$\det(C(\mathbf{p}) - I\boldsymbol{\alpha}) = \begin{vmatrix} -\alpha_1 & 0 & \dots & 0 & -a_0 \\ 1 & -\alpha_2 & \dots & 0 & -a_1 \\ 0 & 1 & \dots & 0 & -a_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & -\alpha_n - a_{n-1} \end{vmatrix} , \tag{2.20}$$

where $\boldsymbol{\alpha}$ is the vector of zeros of $p(z)$ and I is the identity matrix.

Many researchers, from Dario Bini, §4.2 starting on page 27, through Joab Winkler, §4.5 starting on page 34, to Zhonggang Zeng, §4.6 starting on page 37, use algorithms for locating the eigenvalues of $C(\mathbf{p})$ as a mechanism for locating the zeros of $p(z)$.

QR Algorithm

A popular approach taken by researchers for determining the eigenvalues of $C(\mathbf{p})$, and therefore the zeros of $p(z)$, is known as the QR algorithm, first described by John Francis in [Fra61] and [Fra62].

The matrix A is decomposed into a product of an *orthogonal matrix* and an *upper triangular matrix*, the process continuing by iterating. Start with $A_0 = A$. Compute the QR decomposition $A_k = Q_k R_k$, which leads to

$$A_{k+1} = R_k Q_k = Q_k^T Q_k R_k Q_k = Q_k^T A_k Q_k = Q_k^{-1} A_k Q_k \quad , \tag{2.21}$$

so the A_k are *similar* and therefore have the same eigenvalues. The algorithm is *numerically stable* but can use a large number of expensive calculations.

Under suitable conditions [Fra62], the matrices A_k converge to a triangular matrix, called the *Schur form* of A [GL86, pp. 219–226].

Let $p(z)$ be defined as in Equation (2.3) on page 6. For any finite integer $k > 0$,

the matrix of size $(n+k) \times k$, given by

$$C_k(p(z)) = \overbrace{\begin{bmatrix} a_n & & & \\ a_{n-1} & a_n & & \\ \vdots & a_{n-1} & \ddots & \\ a_0 & \vdots & \ddots & a_n \\ & a_0 & \ddots & a_{n-1} \\ & & \ddots & \vdots \\ & & & a_0 \end{bmatrix}}^{k} \quad , \tag{2.22}$$

is the kth order **convolution matrix** associated with $p(z)$.

Let $p(z)$ and $q(z)$ be polynomials of degrees n and m respectively, with $r(z) = p(z)q(z)$. Then $\mathbf{r}(z)$ is the **convolution** of $\mathbf{p}(z)$ and $\mathbf{q}(z)$ defined by

$$\begin{aligned} \mathbf{r}(z) &= \text{conv}(\mathbf{p}(z),\ \mathbf{q}(z)) \quad , \\ &= C_m(p(z))\mathbf{q}(z) \quad , \\ &= C_n(q)\mathbf{p}(z) \quad . \end{aligned} \tag{2.23}$$

Let $p(z)$ be defined as in Equation (2.3) on page 6 and let $p'(z)$ be its derivative. For $k = 1,\ 2,\ \ldots,\ n$, the matrix of size $(n+k) \times (2k+1)$

$$S_k(p(z)) = [\ C_{k+1}(p'(z)) \mid C_k(p(z))\ ] \quad , \tag{2.24}$$

is the kth order **Sylvester discriminant matrix** associated with $p(z)$. Let

$$\begin{aligned} p(z) &= \sum_{i=0}^{n} a_i z^i \quad , \\ q(z) &= \sum_{i=0}^{m} b_i z^i \quad , \end{aligned} \tag{2.25}$$

then the matrix of size $(m+n) \times (m+n)$, given by

$$S(p,q) = \overbrace{\begin{bmatrix} a_n & & & & b_m & & & \\ a_{n-1} & a_n & & & b_{m-1} & b_m & & \\ \vdots & a_{n-1} & \ddots & & \vdots & b_{m-1} & \ddots & \\ a_0 & \vdots & \ddots & a_n & b_0 & \vdots & \ddots & b_m \\ & a_0 & \ddots & a_{n-1} & & b_0 & \ddots & b_{m-1} \\ & & \ddots & \vdots & & & \ddots & \vdots \\ & & & a_0 & & & & b_0 \end{bmatrix}}^{m \qquad\qquad n} \quad , \tag{2.26}$$

is the **Sylvester resultant matrix** associated with $p(z)$ and $q(z)$ [vzGG99, p. 144].

2.2.2 Circular Arithmetic

Many mathematicians from Irene Gargantini and Peter Henrici [GH72] through Miodrag Petković [Pet89] to Monday Ikhile [Ikh02] use circular arithmetic when computing approximations to the zeros of a polynomial.

In particular, a paper by Ljaljana Petković [Pet86, pp. 371–372] investigates the order of evaluation of expressions in circular arithmetic in order to minimise the radii of resultant circles, e.g. inversion should be applied before multiplication, so, for circular disks A, B, C, D we have

$$radius\left(\frac{AB}{CD}\right) \geq radius\left(\frac{A}{C} \times \frac{B}{D}\right) \quad . \tag{2.27}$$

The centre of a circle is an approximation to a zero of a polynomial and the radius is a measure of the error involved. Generally, circular regions are denoted by upper-case letters, $A,\ B,\ \ldots,\ Z$, complex numbers by lower-case letters, $a,\ b,\ \ldots,\ z$, and non-negative real numbers by Greek lower-case letters, $\alpha,\ \beta,\ \ldots,\ \zeta$.

When our circular regions are circles then the region Z with centre c and radius ρ,

$$Z = \{z :\ |z - c| \leq \rho\} \quad , \tag{2.28}$$

is written

$$Z = [c;\ \rho] \quad , \tag{2.29}$$

with $c = \text{mid } Z$ and $\rho = \text{rad } Z$. Thus we have

$$a \pm Z = [a \pm c;\ \rho] \quad , \tag{2.30}$$

$$aZ = [ac;\ |a|\rho] \quad , \tag{2.31}$$

and the sum or difference of two circles is

$$Z_1 \pm Z_2 = [c_1 \pm c_2;\ \rho_1 + \rho_2] \quad , \tag{2.32}$$

while the product of two circles is

$$Z_1 Z_2 = [c_1 c_2;\ |c_1|\rho_2 + |c_2|\rho_1 + \rho_1\rho_2] \quad , \tag{2.33}$$

which is both commutative and associative [GH72, p.308]. The distributive law is replaced by the relation

$$Z_1(Z_2 + Z_3) \subseteq Z_1 Z_2 + Z_1 Z_3 \quad . \tag{2.34}$$

If $0 \notin Z$,

$$\frac{1}{Z} = \left[\frac{\bar{c}}{c\bar{c} - \rho^2};\ \frac{\rho}{c\bar{c} - \rho^2}\right] \quad . \tag{2.35}$$

The square root of a circle has two cases. If $0 \in Z$, then

$$Z^{\frac{1}{2}} = [0;\ (|c| + \rho)^{\frac{1}{2}}] \quad , \tag{2.36}$$

while if $0 \notin Z$, then $Z^{\frac{1}{2}}$ is the union of the two circles

$$\left[\pm c^{\frac{1}{2}};\ \frac{1}{|c|^{\frac{1}{2}} + (|c| - \rho)^{\frac{1}{2}}}\right] \quad , \tag{2.37}$$

where $c^{\frac{1}{2}}$ is one of the two numbers satisfying $(c^{\frac{1}{2}})^2 = c$.

Chapter 3

A Historic Perspective

This chapter introduces a brief history of the people whose early work led to our methods for locating all the zeros of a polynomial and helped those currently working in this area.

3.1 George Collins

In the early 1960s George Collins at IBM and the Computer Science Department of the University of Wisconsin-Madison developed PM [Col66], a system for polynomial manipulation, an early version of which was operational in 1961 with improvements added to the system through to 1966.

In the late 1960s George developed and used the SAC-1 system[1] as the successor of PM for the manipulation of polynomials and rational functions. We quote from the abstract of Professor Collins' paper [Col71, p. 144] describing SAC-1.

> "SAC-1 is a program system for performing operations on multivariate polynomials and rational functions with infinite-precision coefficients. It is programmed, with the exception of a few simple primitives, in ASA Fortran. As a result the system is extremely accessible, portable, easy to learn, and indeed has been implemented at more than 20 institutions.
>
> "The SAC-1 system's range of programmed capabilities is exceptionally broad, including, besides the usual operations, polynomial

[1] Symbolic and Algebraic Computing.

> greatest common divisor and resultant calculation, polynomial factorisation, exact polynomial real zero calculation, partial fraction decomposition, rational function integration, and solution of systems of linear equations with polynomial coefficients.
>
> "SAC-1 is also outstanding in its computing time efficiency, which is achieved partially through the use of appropriate data structures, but primarily through the use of mathematically sophisticated and analysed algorithms, which are briefly surveyed. The efficiency gains, frequently orders of magnitude, are such that many new applications are rendered feasible."

This must have been a remarkable system for its time. The University of Wisconsin in the late 1960s was running a UNIVAC 1108 computer which came originally with a massive (for its time) 65,536 (64Ki) words of memory.

In 1980 Professor Collins introduced SAC-2 as a replacement for SAC-1. It was programmed in the ALDES language, designed by Rüdiger Loos and Professor Collins [Col85].

3.2 Theodorus (Dirk) Dekker

Theodorus is a Dutch mathematician at the University of Amsterdam. At the end of his most important paper, [Dek68, pp. 198–199], he derived the radius of a circle centred at the origin containing all the zeros of $p(z)$, namely

$$|z| = 2\kappa, \quad \kappa = \max_{0}^{n-1} \left| \frac{a_i}{a_n} \right|^{\frac{1}{n-i}} \quad . \tag{3.1}$$

This is one of many such bounds defining a region (usually a circle) containing the zeros of a polynomial, e.g. [Hen74, pp. 457–458], but has proved satisfactory for our use, i.e. there is no real gain in applying all of these, and taking the minimum radius.

Our initial inclusion region is the smallest possible **square** containing the circle given by Equation (3.1) on page 14. This is illustrated in Figure 3.1 on page 15.

The reason for the orientation of the circle is because, when we first started, many of our test polynomials had real zeros. These would lie on a horizontal line through the origin, which could cause problems with inclusion regions having part of that horizontal line as one of its boundaries.

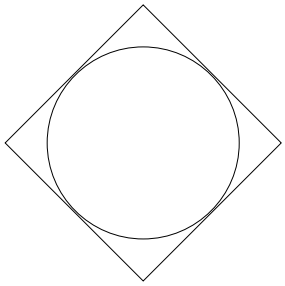

Figure 3.1: First Inclusion Region

3.3 Irene Gargantini

Irene is currently Professor Emeritus, Department of Computer Science at the University of Western Ontario, Canada. Her initial work in numerical analysis included results in approximation theory, continued fractions, and polynomial root-finding. Her paper with Peter Henrici, on *circular arithmetic* [GH72], is regarded as a classic in the field.

REMARK 1. *Paragraph removed. Winkler point 1.*

In [HG69] the authors describe convergent algorithms for computing all the zeros of a polynomial. However, before this was published Irene and Münzner had published a Research Report [GM67] describing a Fortran program for determining all the zeros of a polynomial where the search stage involves squares[2], rather than circles.

In [GH72] the authors

> "... construct (i) a version of Newton's method with error bounds, and (ii) a cubically convergent algorithm for the simultaneous approximations of all zeros of a polynomial."

These error bounds are possible due to the use of circular arithmetic, see §2.2.2 starting on page 11.

3.4 Peter Henrici

Peter was Professor of Mathematics at the University of California in Los Angeles and later at the Swiss Federal Institute of Technology in Zürich. He died in 1987.

[2]This is our approach as well.

He wrote eleven books and over 80 research papers, some with Irene Gargantini, see §3.3. He was one of the first researchers to investigate *circular arithmetic* as a tool for locating polynomial zeros [Hen71]. He collaborated in this work with Irene Gargantini [GH72].

His high point was the three-volume work *Applied and Computational Complex Analysis*, of which the first volume [Hen74] dealt with the zeros of polynomials (amongst other things). Many of the theorems by various authors quoted in this thesis are summarised in his book. The following two theorems from his book are stated using our notation and definitions.

Theorem 3.1. *Let $p(z)$ be defined as in Equation (2.3). All zeros of $p(z)$ lie outside the closed unit circle $|z| \leq 1$ if and only if*

$$\gamma_i > 0, \quad i = 1,\ 2,\ \ldots,\ n \quad . \tag{3.2}$$

The calculation of the numbers γ_i by using Equation (2.18) is called the **Schur-Cohn algorithm** [Hen74, p. 494]. Under the additional hypothesis that all $\gamma_i \neq 0$, the algorithm can be used to determine the exact number of zeros in the unit circle.

Theorem 3.2. *Let $p(z)$ be defined as in Equation (2.3). Let the numbers γ_i defined by Equation (2.18) satisfy $\gamma_i \neq 0$, $i = 1,\ 2,\ \ldots,\ n$. If those indices i for which $\gamma_i < 0$ are denoted by k_i, $i = 1,\ 2,\ \ldots,\ m$, where $k_1 < k_2 < \cdots < k_m$, then the number $h(p)$ of zeros inside the unit circle is given by*

$$h(p) = \sum_{i=1}^{m} (-1)^{i-1}(n + 1 - k_i) \quad . \tag{3.3}$$

We note that Theorem 3.2 cannot be used to calculate the number of zeros inside the unit circle if some of the γ_i are zero. For further details about this topic see [Hen74, p. 496].

3.5 Alston Householder

Alston was born in 1904 and died in 1993. He was Head of the Mathematics Panel of the Oak Ridge National Laboratory from 1948 to 1969 when he became Professor of Mathematics at the University of Tennessee. He retired in 1973.

He derived a family of IFs based on the poles of a function known as Householder's method. Note that in Equation (3.4) $[f]^{(d)}$ is the dth derivative of f and not a power, so for simple zeros only and, using our notation, the family of

IFs

$$\hat{z} = z + (\rho - 1)\frac{\left[\frac{1}{p(z)}\right]^{(\rho-2)}}{\left[\frac{1}{p(z)}\right]^{(\rho-1)}} \quad , \ \rho = 2,\ 3,\ \ldots, \tag{3.4}$$

comprises those found in Table 3.1 on page 17.

Order	Name	Equation	Page
2	Newton	(5.13)	50
3	Halley	(5.14)	50
4	Kiss	(B.3)	121

Table 3.1: Family of Simple IFs

The original derivation of Equation (3.4) is found in Alston's book [Hou70, p. 169].

3.6 Misako Ishiguro

Misako was one of the first researchers to solve polynomials with multiple zeros using the $\gcd(p(z), p'(z))$ method of deflating $p(z)$ into factors with simple zeros [Ish72]. This technique has been refined by Joab Winkler, see §4.5 on page 34.

3.7 Peter Jarratt

In 1975 Peter was appointed to the dual posts of Director of the Computer Centre and Professor of Computing at the University of Birmingham. He retired in September 2000.

His methods, such as [Jar66a] and [Jar69], are experiencing a revival with modern authors such as Ramandeep Behl et al. [BKS12], Jisheng Kon and Yitian Li [KL07], J R Sharma [Sha07], and Miodrag Petković, see §4.4 on page 28.

3.8 Jean-Louis Lagouanelle

Jean-Louis is a researcher at the Institut de Recherche en Informatique de Toulouse, France. In [Lag66, p. 627] he derived the following formula concerning

the multiplicity m_i of the zero α_i of $p(z)$

$$m_i = \lim_{z_i \to \alpha_i} \left| \frac{1}{u'(z_i)} \right|, \quad i = 1,\ 2,\ \ldots,\ N, \tag{3.5}$$

where $u(z)$ is defined as in Equation (2.6).

Our algorithm compares this value (computed at the centre of an inclusion region) with Marden's count (Theorem 3.6 on page 21) of the number of zeros in the inclusion region. If these are identical for each inclusion region, then we can complete the search stage and move on to the iterative stage.

3.9 Derrick (Dick) Lehmer

Dick was instrumental in applying the use of early computers to problems that were virtually impossible to tackle by hand, especially sieve methods from number theory. He died in 1991.

He first described the Lehmer-Schur algorithm [Leh61] for finding numerical approximations to the zeros of a polynomial.

The Lehmer-Schur algorithm is based on a theorem by Issai Schur [Mar66, p. 198] for answering the *basic question*

"Does the polynomial $p(z)$ have a zero inside the circle $|z - c| = \rho$?"

Algorithm 3.1. *The algorithm consists of the following three steps.*

1. *Assuming that $p(0) \neq 0$, start with the unit circle*[3] *asking the basic question, doubling (or halving) the radius at each step, to find an annulus*

$$R < |z| < 2R \tag{3.6}$$

where R is a power of two containing at least one zero of $p(z)$, while the inner circle contains no zeros. This annulus can be covered by eight *overlapping circles of radius $\frac{5R}{6}$ with centres at $\frac{5R}{3}\exp(\frac{\pi i k}{4})$, for $k = 0,\ 1,\ \ldots,\ 7$.*

Use the basic question on each one of these circles in turn and find the first circle containing at least one zero of $p(z)$. Call the centre of this circle β_1. Continuing the process by halving the radius each time eventually yields an annulus

$$R_1 < |z - \beta_1| < 2R_1 \tag{3.7}$$

[3] It would probably be better to start with Dirk Dekker's circle, Equation (3.1) on page 14, because this circle contains all the zeros.

which completes Step 1.

2. *Cover this annulus with another eight circles to find the first one containing at least one zero. Call the centre of this circle β_2. Continuing the process by halving the radius each time yields an annulus*

$$R_2 < |z - \beta_2| < 2R_2 \tag{3.8}$$

which completes Step 2.

Figure 3.2 shows the result after Step 2. Dashed circles have no zeros in them. Note that a zero in a given annulus may not be the final zero that is

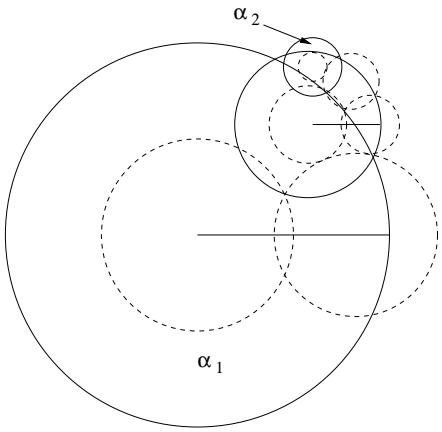

Figure 3.2: Lehmer-Schur in Action

used. For example, in Figure 3.2, the original annulus contains the zero marked α_1. However, the algorithm is homing in on the zero marked α_2 which was not in the original annulus.

3. *Repeat Step 2 until the required accuracy is obtained.*

3.9.1 The Basic Question

By a linear transformation the circle $|z - c| = \rho$ can be replaced by the unit circle. The polynomial

$$r(z) = p(\rho z + c) \tag{3.9}$$

has a zero

$$\beta = (\alpha - c)/\rho \tag{3.10}$$

for every zero α of p and $|\beta| < 1$ if and only if $|\alpha - c| < \rho$. The question is now

"Does $p(z)$ have a zero inside the unit circle?"

This question is answered by using the iterated Schur transforms given by Equation (2.15) on page 8 as the basis of the following theorem, stated using our notation and definitions.

Theorem 3.3. *Let $p(z)$, defined as in Equation (2.3) on page 6, have no zero on the unit circle. Suppose $p(0) \neq 0$. Let γ_i be defined as in Equation (2.18) on page 8. If, for some $i > 0$, $\gamma_i < 0$, then p has at least one zero inside the unit circle. Alternatively, if $\gamma_i > 0$ for $1 \leq i < k$ and $T^{k-1}(p(z))$ is a constant, then no zero of $p(z)$ lies inside the unit circle.*

The hypothesis that $p(z)$ has no zero on the unit circle is dealt with by the following theorem, again stated using our notation and definitions.

Theorem 3.4. *Theorem 3.3 remains true if we weaken its hypothesis by deleting the first sentence.*

However, this still leaves the situation when $T^{k-1}(p(z))$ is not a constant. Professor Lehmer [Leh61, p. 158] suggests that the radius of the circle be increased, a factor of $\frac{3}{2}$ is mentioned, in this case.

3.10 Morris Marden

Morris is credited with founding the Department of Mathematics at the University of Wisconsin-Milwaukee as a research department. He retired in 1975 after spending 45 years teaching in Milwaukee. He died in 1991.

His seminal work *Geometry of Polynomials* [Mar66] contains some intriguing results.

Let $p(z)$ be defined as in Equation (2.3) on page 6 and let $p^*(z)$ be defined as in Equation (2.14) on page 7. Let the sequence of polynomials $p_i(z)$ be defined by the recursive formula

$$p_{i+1}(z) = \bar{a}_0^{(i)} p_i(z) - a_n^{(i)} p_i^*(z), \quad i = 0,\ 1,\ \ldots,\ n-1 \tag{3.11}$$

with $p_0(z) = p(z)$ [Mar66, p. 195]. Thus $p_i(z)$ is $T^i(p(z))$ as defined in Equation (2.15) on page 8. Finally define

$$P_i = \gamma_1 \gamma_2 \ldots \gamma_i, \quad i = 1,\ 2,\ \ldots,\ n \tag{3.12}$$

which leads into the following theorem stated using our notation and definitions.

Theorem 3.5. *Let $p(z)$ be defined as in Equation (2.3) on page 6, let $p^*(z)$ be defined as in Equation (2.14) on page 7, let $T^i(p(z))$ be defined as in Equation (2.15) on page 8, and let γ_i be defined as in Equation (2.18) on page 8. If*

p of the products P_i defined by Equation (3.12) on page 20 are negative and the remaining $n - p$ are positive, then $p(z)$ has p zeros in the unit circle, no zeros on the unit circle, and $n - p$ zeros outside the unit circle.

Under the hypothesis of Theorem 3.5 on page 20, no $T^i(p(z)), \quad i \leq n$, may be identically zero. The following theorem covers that case [Mar66, p. 203]. Again, it is stated using our notation and definitions.

Theorem 3.6. *If for some $k < n$, $P_k \neq 0$ in Equation (3.12) on page 20 but $T^{k+1}(p(z)) \equiv 0$, then $p(z)$ has $n - k$ zeros on or symmmetric in the unit circle at the zeros of $T^k(p(z))$. If p of the P_i, $\quad i = 1,\ 2,\ \ldots,\ k$, are negative, $p(z)$ has p additional zeros inside the unit circle and $k - p$ additional zeros outside the unit circle.*

This theorem is the workhorse of our search stage. For each of our square inclusion regions, we compute the number of zeros in the smallest enclosing circle.

Some zeros might be counted twice, due to the overlap of enclosing circles. This problem is dealt with in the consolidation phase of the search stage. See §5.2.2, starting on page 47.

3.11 Alexander Ostrowski

Alexander was one of the first researchers to investigate solutions to the following question [Ost66].

> "Given two polynomials of equal degree and having coefficients that are nearly equal, by how much can their zeros differ?"

Alston Householder covers this topic in his book [Hou70, pp.74–81]. Paraphrasing his account, using our notation, if $p(z)$ is defined as in Equation (2.4) on page 6, and

$$q(z) = z^n + b_{n-1}z^{n-1} + \ldots + b_0, = \prod_i (z - \beta_i)^{m_i} \quad , \tag{3.13}$$

with

$$\gamma = 2 \max_i (|a_i|^{\frac{1}{i}}, |b_i|^{\frac{1}{i}}) \quad , \tag{3.14}$$

and

$$\epsilon^n = \sum_i |b_i - a_i| \gamma^{n-i} \quad , \tag{3.15}$$

then the zeros α_i and β_i of $p(z)$ and $q(z)$, respectively, can be ordered so that

$$|\alpha_i - \beta_i| < 2n\epsilon \quad . \tag{3.16}$$

This result is especially interesting because it is an early attempt to quantify how perturbations in the coefficients of a polynomial affect the values of the zeros of the perturbated polynomial.

Coming up-to-date, there are researchers such as Joab Winkler (see §4.5 starting on page 34) and Zhonggang Zeng (see §4.6 starting on page 37) investigating the behaviour of zeros of a polynomial on a **pejorative manifold** as the coefficients of the polynomial are perturbed [Kah72].

This is just one aspect of the influence Alexander Ostrowski has had on many branches of mathematics.

3.12 James Pinkert

James is Professor Emeritus in the Department of Computer Science at the University of Wisconsin-Madison. He was a member of the team led by Professor George Collins that developed and used the SAC-1 system, see §3.1.

In an early paper [Pin76] based on his PhD thesis, Professor Pinkert describes an exact algorithm for locating the zeros of a polynomial using the SAC-1 system. He stores the coefficients of $p(z)$ as rational complex numbers, so that all calculations are exact. His search stage is based on rectangles in the complex plane.

If the multiplicities are not important, he applies the algorithm to $\dfrac{p}{\gcd(p,\ p')}$, which has only simple zeros. If the multiplicities are needed, he factorises $p(z)$ as follows

$$p(z) = \omega_1(z)\omega_2^2(z)\cdots\omega_M^M(z) \tag{3.17}$$

where each $\omega_i(z)$ has only simple zeros. This is Joab Winkler's approach, see §4.5, starting on page 34.

3.13 Marica Prešić

Marica is a professor in the Mathematical Institution at the University of Belgrade, Serbia working in the fields of mathematical linguistics, universal algebra, and mathematical logic. She is the widow of Slaviša (Bogomira) Prešić, one of

the founders of modern logic in Serbia (together with Alexander Kron). Slaviša died in 2008.

REMARK 2. *Paragraph removed. Winkler point 1.*

One of her early papers [Pre71][4] was the defining moment that led to our research into IFs and ultimately searching the complex plane for regions containing zeros of a polynomial to complement that work.

Her IF is a modification of the rational form of the simple second-order one-point IF given in Equation (B.5) on page 121 dealing with a subset of zeros $z_1,\ z_2,\ \ldots,\ z_k$, namely (using our notation)

$$\hat{z}_\nu = z_\nu - \frac{p(z_\nu)}{\prod_{i\neq\nu}^{k}(z_\nu - z_i)p_{\mathbf{z}}(z_\nu)} \quad , \quad \nu = 1,\ 2,\ \ldots,\ k \quad , \tag{3.18}$$

where

$$p_{\mathbf{z}}(z_\nu) = \frac{p(z_\nu)}{\prod_{i\neq\nu}(z_\nu - z_i)} \quad . \tag{3.19}$$

3.14 Ernst Schröder

According to Joseph Traub [Tra64, p. 127] Ernst originally derived what we refer to as Rall's modified Newton IF, given as Equation (5.36) on page 57, in the 19th century [Sch70, p. 352].

Some of his earlier work on IFs is brought up to date using circular arithmetic, see §2.2.2 on page 11, in a paper by Miodrag Petković [Pet90].

Just a short digression. Ernst is probably better known for two quantities that bear his name.

Schröder's Equation

Given the function $f(x)$, find the function $\Psi(x)$ that satisfies

$$\Psi(f(x)) = s\Psi(x) \quad , \tag{3.20}$$

where

$$0 < s < 1 \quad . \tag{3.21}$$

See [Kuc64] for further details.

[4]The title, in English, is: An Iterative Process for Determining k Zeros of a Polynomial.

Schröder's Number

The number S_n describes the number of paths from the south-west corner $(0,\ 0)$ of an $n \times n$ grid to the north-east corner $(n,\ n)$ using only single steps north, north-east, or east, that do not rise above the SW-NE diagonal. The S_n are generated by the recurrence relation

$$S_n = S_{n-1} + \sum_{i=0}^{n-1} S_i S_{n-1-i} \quad . \tag{3.22}$$

3.15 Gilbert (Pete) Stewart

Pete is a Distinguished University Professor Emeritus in the Department of Computer Science at the Institute for Advanced Computer Studies, University of Maryland .

In [Ste69] he states that Lehmer's method [Leh61] is numerically unstable within the algorithm for determining if $p(z)$ has a zero in a circle.

First note that $p(z)$ has a zero in $|z - c| \leq \rho$ if and only if

$$r(z) = p(\rho z + c) \tag{3.23}$$

has a zero in the unit circle [Ste69, p. 831].

Algorithm 3.2. *The algorithm consists of the following three steps.*

1. *Calculate the coefficients of*

$$q(z) = b_n z^n + b_{n-1} z^{n-1} + \cdots + b_0 = p(z + c) \tag{3.24}$$

2. *Calculate the coefficients of*

$$r(z) = c_n z^n + c_{n-1} z^{n-1} + \cdots + c_0 = q(\rho z) \tag{3.25}$$

3. *Determine whether $r(z)$ has a zero inside the unit circle.*

The scaling step, Equation (3.25), may cause problems. The coefficients of $r(z)$ are given by

$$c_i = \rho^i b_i \quad , i = 0,\ 1, \ldots,\ n \quad . \tag{3.26}$$

If n is large and $\rho > 1$, the absolute values of the coefficients may also overflow. Similarly, if $\rho < 1$, the absolute values of the coefficients may underflow.

Professor Stewart suggests a scaling algorithm for overcoming these problems, which we now present.

Algorithm 3.3. *Let Ω be the largest positive number that can be represented in the computer. Then a set of coefficients, different from those in Equation (3.26), are calculated as follows.*

1. *Determine the largest number σ satisfying*

$$\sigma|b_i| \leq \Omega, \quad i = 0,\ 1,\ \ldots,\ n \quad . \tag{3.27}$$

2. *If $\rho < 1$, set*

$$c_i = (\sigma\rho^i)b_i, \quad i = 0,\ 1,\ \ldots,\ n \tag{3.28}$$

   where $c_i = 0$ if underflow occurs in the computation.

3. *If $\rho > 1$, set*

$$c_i = (\sigma\rho^{i-n})b_i, \quad i = n,\ n-1,\ \ldots,\ 0 \tag{3.29}$$

   where c_i equals 0 if underflow occurs in the computation.

Interestingly, our implementation uses Algorithm 3.2 rather than Algorithm 3.3 in our search stage because the GNU Multiple Precision Arithmetic Library [Gra11] has such a large Ω.

Professor Stewart also has a different version of the algorithm for determining whether a polynomial has a zero in the unit circle.

Theorem 3.7. *Let $p(z)$ be defined as in Equation (2.3) on page 6 and let $p^*(z)$ be defined as in Equation (2.14) on page 7. Let $m = \dfrac{a_n}{\bar{a}_0}$. Then, if $m \geq 1$, $p(z)$ has a zero in the unit circle. However, if $m < 1$, the polynomial*

$$p_1(z) = p(z) - mp^*(z) \tag{3.30}$$

is of degree less than n and has the same number of zeros inside the unit circle as $p(z)$. Moreover, $p_1(0) \neq 0$.

3.16 Joseph Traub

Joseph was born in 1932. He is currently Edwin Howard Armstrong Professor of Computer Science at Columbia University. He previously held positions at Bell Laboratories, University of Washington, and Carnegie Mellon University. While in the Research Division at Bell Laboratories he worked in the field of computational complexity, applying his knowledge to the solutions of non-linear

equations. This culminated in his book *Iterative Methods for the Solution of Equations* [Tra64] which is still in print.

In 1966 he spent a sabbatical at Stanford University where he worked with another computer scientist, Michael Jenkins, to formulate the *Jenkins-Traub Algorithm for Polynomial Zeros* [JT70] which was first published in Michael Jenkins' PhD thesis [Jen69]. The algorithm itself, written in Fortran, was finally published in 1972 [JT72].

During our research we have made extensive use of Traub's seminal work [Tra64], especially his derivations of the single-point IFs for simple zeros, [Tra64, pp. 78–104], and his derivations of the single-point IFs for multiple zeros, [Tra64, pp. 126–157].

His exhaustive collection of references was invaluable in enabling us to check back on who first defined various IFs. Any errors in this respect are, of course, entirely ours.

Chapter 4

Others Working in the Same Field

This chapter introduces a number of other people who are currently researching in the area of the computation of polynomial zeros.

4.1 Ramandeep Behl et al.

Ramandeep and his co-researchers have published a paper describing improvements to Jarratt's method [BKS12] using multipoint IFs, which are not the thrust of our approach.

However, his approach of parameterising the IFs and then choosing the appropriate parameter values in order to optimise the IFs in terms of simplicity or order of convergence (or both) is of increasing interest to other researchers, e.g. Young Kim [Kim12].

Obviously, a research topic to keep an eye on for the future.

4.2 Dario Bini and Giuseppe Florentino

Dario is currently a full professor of Numerical Analysis in the Departent of Mathematics at the University of Pisa, Italy.

He and Giuseppe have investigated the connection between the zeros of $p(z)$, as defined in Equation (2.3) on page 6, and the eigenvalues of its associated

companion matrix $C(p)$, as defined in Equation (2.19) on page 8. This is also the approach taken by Joab Winkler, §4.5 starting on page 34, and Zhonggang Zeng, §4.6 starting on page 37.

One excellent result of Dario's work is an algorithm for computing the zeros of a polynomial [BF00a] using the GNU Multiple Precision Arithmetic Library [Gra11] as we do. A detailed description of the algorithm is found in [BF00b] together with an exhaustive set of test polynomials[1].

One of his current research interests is using the QR eigenvalue algorithm for companion matrices [BBE+10].

4.3 Stef Graillat et al.

Stef and his research group are working on improving the accuracy of the results from applying Horner's algorithm for computing the value of $p(z)$, using fixed-length arithmetic [GLL09].

Their paper has provided one polynomial with two zeros, of multiplicities five and eleven respectively, which were easily solved by our algorithm with minimal tweaking, see §6.12 starting on page 72.

4.4 Miodrag Petković et al.

Miodrag was born in 1948. He is currently a full professor in the Department of Mathematics, Faculty of Electronic Engineering at the University of Niš in Niš, Serbia. His research interests are the solutions of polynomial equations using interval arithmetic or circular arithmetic. His PhD thesis [Pet80], awarded by the University of Niš, was an analysis of iterative interval methods for solving equations.

Interestingly, he is one of the few researchers who acknowledge the use of rectangular regions (in the complex plane) as well as circles [Pet89, p. 18] for interval arithmetic. Others include Krishnamurthy and Venkateswaren [KV81] and Herbert Wilf [Wil78]. In passing, he states that we (i.e. Mick Farmer and George Loizou in [FL77]) used Marden's inclusion test [Mar66, pp. 194–197], but finishes by estimating the multiplicities of the zeros by using Lagouanelle's limiting

[1] Up to degree 6,400!

formula [Lag66, p. 627][2]

$$m_\nu = \lim_{z_\nu \to \alpha_\nu} \frac{p'(z_\nu)^2}{p'(z_\nu)^2 - p(z_\nu)p''(z_\nu)} \quad , \tag{4.1}$$

which can be written more succinctly using our definition given in Equation (2.6) on page 6 as

$$m_\nu = \lim_{z_\nu \to \alpha_\nu} \frac{1}{u'(z_\nu)} \quad . \tag{4.2}$$

In order to obtain a precise estimate of the multiplicities, Miodrag suggests using an improved approximation to z_ν, obtained by applying Newton's formula

$$\hat{z}_\nu = z_\nu - \frac{p(z_\nu)}{p'(z_\nu)} \quad . \tag{4.3}$$

He notes that this formula has only linear convergence if the multiplicity is greater than one. To overcome this, he suggests applying a modified form of Newton's formula

$$\hat{z}_\nu = z_\nu - \frac{\dfrac{p(z_\nu)}{p'(z_\nu)}}{1 - \dfrac{p(z_\nu)p''(z_\nu)}{p'(z_\nu)^2}} \quad , \tag{4.4}$$

which has second-order convergence. Again, using the definition given in Equation (2.6) on page 6, this equation can be written more succinctly as

$$\hat{z}_\nu = z_\nu - \frac{u(z_\nu)}{u'(z_\nu)} \quad . \tag{4.5}$$

He suggests stopping the search stage either when Marden's inclusion test yields the correct number of zeros or when Lagouanelle's limiting formula yields the correct number of zeros. This differs from our approach, which is to apply both tests until they agree with one another.

§2.2 of [Pet89] describes the complex interval (circular) arithmetic that he uses for programming his iterative methods. In particular, this addresses the major problem with circular arithmetic, namely that some operations result in a new circular region with a larger radius than that of their arguments.

In Chapter 3 of [Pet89], Miodrag considers the second-order IF first presented by Weierstrass [Wei03],

$$\hat{z}_\nu = z_\nu - \frac{p(z_\nu)}{\prod_{i \neq \nu}(z_\nu - z_i)}, \quad \nu = 1,\ 2,\ \ldots .\ n \quad , \tag{4.6}$$

[2]The value of the formula is rounded to the nearest integer.

and subsequently derived by other researchers, including ourselves [FL75]. Under certain conditions he shows that the IF converges in the sense that a zero is contained in each circular region and that the radii of the circular regions monotonically tend to zero. In the case of simple zeros only that

$$\max_{\nu} \hat{\epsilon}_{\nu} = \lambda(n) \max_{\nu} \epsilon_{\nu}^{2} \quad , \tag{4.7}$$

where $\lambda(n)$ is dependent solely on the degree of the polynomial, n.

We refer to Equation (4.6) on page 29 as being performed in a *parallel* fashion. Improved convergence can be obtained if the new approximations are computed in a *serial* fashion[3] in which the newly computed approximations are used as soon as they become available. Miodrag refers to this as the single-step method

$$\hat{z}_{\nu} = z_{\nu} - \frac{p(z_{\nu})}{\prod_{i<\nu}(z_{\nu} - \hat{z}_{i}) \prod_{i>\nu}(z_{\nu} - z_{i})}, \quad \nu = 1,\ 2,\ \ldots .\ n \quad , \tag{4.8}$$

which has also been analysed by Alefeld and Hertzberger [AH74].

§3.2 of [Pet89] defines a third order IF based on the Lagrangian interpolation of $p(z)$ for the points $z_1,\ z_2,\ \ldots,\ z_n$, that is

$$p(z) = \sum_{i} \frac{q(z)}{q'(z_i)(z - z_i)} p(z_i) + q(z) \quad , \tag{4.9}$$

where

$$q(z) = (z - z_1)(z - z_2) \cdots (z - z_n) \quad . \tag{4.10}$$

Using the abbreviation

$$W_i = \frac{p(z_i)}{\prod_{j \neq i}(z_i - z_j)} \quad , \tag{4.11}$$

based on the Weierstrass correction given in Equation (4.6) on page 29, then the IF

$$\hat{z}_{\nu} = z_{\nu} - \frac{W_{\nu}}{1 - \sum_{i \neq \nu} \frac{W_i}{z_i - z_{\nu}}} \tag{4.12}$$

is of third order. Once again, for simple zeros only and sufficiently close starting approximations to those zeros, this IF converges monotonically. This IF does not appear in previous literature.

It is important to remember that we are extrapolating from Miodrag's IFs ex-

[3] Often referred to as the Gauss-Seidel approach.

pressed in terms of circular arithmetic into our terminology. The centre of one of his circles might not be identical to one of our simple approximations.

In [Pet81] and [Pet89, pp. 83–88] Miodrag defines a family of IFs in complex arithmetic of the form

$$\hat{z}_\nu = z_\nu - \frac{1}{\left[h_k(z_\nu) - \sum_{j\neq\nu} \frac{1}{(z_\nu - z_j)^k}\right]^{\frac{1}{k}}} \quad , \tag{4.13}$$

with order of convergence $k+2$ where

$$h_i(z) = \frac{(-1)^{i-1}}{(i-1)!}\frac{d^i}{dz^i}[\log p(z)] \quad , \quad i = 1,\ 2,\ \ldots \quad . \tag{4.14}$$

We note that this equation comes out of the blue in [Pet89, p. 70]. His notation seems *upside down*. We are happier with

$$h_i(z) = \sum_j \frac{1}{(z-\alpha_j)^i} \tag{4.15}$$

yielding by induction

$$h_{i+1}(z) = \frac{h_i'(z)}{-i} \quad , \ i = 1,\ 2,\ \ldots \tag{4.16}$$

The first two of these are

$$h_1(z) = \frac{1}{u(z)} \tag{4.17}$$

and

$$h_2(z) = \frac{u'(z)}{u(z)^2} \quad . \tag{4.18}$$

The order of convergence of these IFs is $k+2$. For $k=1$, we obtain the third-order IF

$$\hat{z}_\nu = z_\nu - \frac{1}{\frac{p'(z_\nu)}{p(z_\nu)} - \sum_{i\neq\nu}\frac{1}{z_\nu - z_i}} \quad , \ \nu = 1,\ 2,\ \ldots,\ n \tag{4.19}$$

first derived by Hans Maehly [Mae54]. Using the definition given in Equation (2.6) on page 6 we derived this IF [FL75, p. 252] as

$$\hat{z}_\nu = z_\nu - \frac{u(z_\nu)}{1 - u(z_\nu)\sum_{i\neq\nu}\frac{1}{z_\nu - z_i}} \quad , \ \nu = 1,\ 2,\ \ldots,\ n \quad , \tag{4.20}$$

which we stated had been discovered by Kiril Dochev and Byrnev [DB64].

In Equation (4.19) on page 31, Miodrag suggests using Newton's approximation $z_i - u(z_i)$ instead of z_i, to increase the rate of convergence [Pet89, pp. 83–84], namely (using our notation and terminology)

$$\hat{z}_\nu = z_\nu - \frac{u(z_\nu)}{1 - u(z_\nu) \sum_{i \neq \nu} \frac{1}{z_\nu - z_i + u(z_i)}} \quad , \ \nu = 1,\ 2,\ \ldots,\ n \quad . \tag{4.21}$$

This is now an IF of fourth order as shown by Abdel Anourein [Ano77, pp. 244–245].

The parallel method given in Equation (4.20) on page 31 can be accelerated by applying the serial method

$$\hat{z}_\nu = z_\nu - \frac{u(z_\nu)}{1 - u(z_\nu) \left[\begin{array}{l} \displaystyle\sum_{i<\nu} \frac{1}{z_\nu - \hat{z}_i} \\ \displaystyle + \sum_{i>\nu} \frac{1}{z_\nu - z_i} \end{array} \right]} \quad , \nu = 1,\ 2,\ \ldots,\ n \quad . \tag{4.22}$$

The same goes for Equation (4.21) on page 32 as

$$\hat{z}_\nu = z_\nu - \frac{u(z_\nu)}{1 - u(z_\nu) \left[\begin{array}{l} \displaystyle\sum_{i<\nu} \frac{1}{z_\nu - \hat{z}_i + u(\hat{z}_i)} \\ \displaystyle + \sum_{i>\nu} \frac{1}{z_\nu - z_i + u(z_i)} \end{array} \right]} \quad , \nu = 1,\ 2,\ \ldots,\ n \quad , \tag{4.23}$$

though this is computationally more intensive because of the need to compute $u(\hat{z}_i)$.

Götz Alefeld and Jürgen Herzberger proved [AH74] that the **R-order convergence** of Equation (4.22) on page 32 is bounded below by the quantity $2 + \sigma_n$, where $\sigma_n > 1$ is the unique positive zero of the equation

$$p_n(\sigma) = \sigma^n - \sigma - 2 \quad . \tag{4.24}$$

Gradimir Milovanović and Miodrag proved [MP83] that the R-order convergence of Equation (4.23) on page 32 is bounded below by the quantity $2(1+\sigma_n)$, where $\sigma_n > 1$ is the unique positive zero of the equation

$$p_n(\sigma) = \sigma^n - \sigma - 1 \quad . \tag{4.25}$$

It is worth re-iterating at this point that the stated order of convergence of the above IFs is correct only for polynomials with simple zeros.

In §4.4 of [Pet89] Miodrag turns his attention to polynomials with multiple zeros. Whilst most of the analysis is concerned with methods using circular arithmetic, he does mention some single-point IFs. For example, the multiple zero version of Equation (4.19) on page 31 is

$$\hat{z}_\nu = z_\nu - \frac{m_\nu}{\dfrac{p'(z_\nu)}{p(z_\nu)} - \displaystyle\sum_{i \neq \nu} \frac{m_i}{z_\nu - z_i}} \quad , \; \nu = 1,\ 2,\ \ldots,\ N \tag{4.26}$$

which we expressed as

$$\hat{z}_\nu = z_\nu - \frac{m_\nu u(z_\nu)}{1 - u(z_\nu) \displaystyle\sum_{i \neq \nu} \frac{m_i}{z_\nu - z_i}} \quad , \; \nu = 1,\ 2,\ \ldots,\ N \tag{4.27}$$

in [FL77] and which is the multiple zero version of Edmond Halley's IF [Hal94]. This parallel method can be accelerated by applying the serial method

$$\hat{z}_\nu = z_\nu - \frac{m_\nu u(z_\nu)}{1 - u(z_\nu) \left[\begin{array}{l} \displaystyle\sum_{i<\nu} \frac{m_i}{z_\nu - \hat{z}_i} \\ + \displaystyle\sum_{i>\nu} \frac{m_i}{z_\nu - z_i} \end{array} \right]} \quad , \; \nu = 1,\ 2,\ \ldots,\ N \quad , \tag{4.28}$$

and, once again, Equations (4.27) and (4.28) can be accelerated further using Newton's approximation instead of z_i yielding

$$\hat{z}_\nu = z_\nu - \frac{m_\nu u(z_\nu)}{1 - u(z_\nu) \displaystyle\sum_{i \neq \nu} \frac{m_i}{z_\nu - z_i + u(z_i)}} \quad , \; \nu = 1,\ 2,\ \ldots,\ N \quad , \tag{4.29}$$

and

$$\hat{z}_\nu = z_\nu - \frac{m_\nu u(z_\nu)}{1 - u(z_\nu) \left[\begin{array}{l} \displaystyle\sum_{i<\nu} \frac{m_i}{z_\nu - \hat{z}_i + u(\hat{z}_i)} \\ + \displaystyle\sum_{i>\nu} \frac{m_i}{z_\nu - z_i + u(z_i)} \end{array} \right]} \quad , \; \nu = 1,\ 2,\ \ldots,\ N \quad , \tag{4.30}$$

for the serial method.

In one of his later papers [PMP10] Miodrag uses higher-order IFs to increase the rate of convergence. He uses a multipoint IF described by Peter Jarratt [Jar66b]

and reintroduced by Ramandeep Behl et al. [BKS12, p. 409], namely (using our notation)

$$\hat{z} = z - \frac{1}{2}u(z)\left[\frac{3p'(y) + p'(z)}{3p'(y) - p'(z)}\right] \quad , \tag{4.31}$$

where

$$y = z - \frac{2}{3}u(z) \quad , \tag{4.32}$$

whose order of convergence is 4 for simple zeros. Substituting this $\hat{z}$ for z_i in Equation (4.19) on page 31 or Equation (4.20) on page 31 yields a new IF whose order of convergence is 6.

4.5 Joab Winkler et al.

Joab is a Reader in the Department of Computer Science at the University of Sheffield.

Like Zhonggang Zeng his approach involves finding the GCD of $p(z)$ and $p'(z)$, i.e. $\gcd(p,\ p')$, many times over, as described in [Win07, pp. 34–40] and recently in [Ver11]. Note that this is only required if $p(z)$ has at least one multiple zero.

Let $p(z)$ be a *monic* polynomial as defined in Equation (2.4) on page 6, and let $\omega_i(z)$ be the product of all factors of degree i in $p(z)$. Then, using our notation

$$p(z) = \omega_1(z)\omega_2^2(z)\cdots\omega_N^N(z) \quad , \tag{4.33}$$

and thus, with $g_0(z) = p(z)$, we define the sequence

$$\begin{aligned}
g_1(z) &= \gcd(g_0(z),\ g_0'(z)) = \omega_2(z)\omega_3^2(z)\cdots\omega_N^{N-1}(z) \quad , \\
g_2(z) &= \gcd(g_1(z),\ g_1'(z)) = \omega_3(z)\omega_4^2(z)\cdots\omega_N^{N-2}(z) \quad , \\
g_3(z) &= \gcd(g_2(z),\ g_2'(z)) = \omega_4(z)\omega_5^2(z)\cdots\omega_N^{N-3}(z) \quad , \\
&\cdots \quad , \\
g_N(z) &= 1 \quad .
\end{aligned} \tag{4.34}$$

A sequence of square-free polynomials $f_i(z),\ i = 1,\ 2,\ \cdots,\ N$, is defined such

that

$$\begin{aligned} f_1(z) &= \frac{g_0(z)}{g_1(z)} = \omega_1(z)\omega_2(z)\cdots\omega_N(z) \quad , \\ f_2(z) &= \frac{g_1(z)}{g_2(z)} = \omega_2(z)\omega_3(z)\cdots\omega_N(z) \quad , \\ &\cdots \quad , \\ f_N(z) &= \frac{g_{N-1}(z)}{g_N(z)} = \omega_N(z) \end{aligned} \tag{4.35}$$

and the products $\omega_i(z)$ are determined from

$$\begin{aligned} \omega_1(z) &= \frac{f_1(z)}{f_2(z)} \quad , \\ \omega_2(z) &= \frac{f_2(z)}{f_3(z)} \quad , \\ &\cdots \quad , \\ \omega_N(z) &= f_N(z) \quad . \end{aligned} \tag{4.36}$$

The equations

$$\begin{aligned} \omega_1(z) &= 0 \quad , \\ \omega_2(z) &= 0 \quad , \\ &\cdots \quad , \\ \omega_N(z) &= 0 \quad , \end{aligned} \tag{4.37}$$

contain only simple zeros, and yield the simple, double, triple, etc. zeros of $p(z)$.

Given that these equations have only simple zeros, they can be solved by some other means, e.g. `roots` in Matlab [Mat12]. Note that this does not guarantee easy sailing from this point. Joab and his researchers in [WLH12, pp. 3480–3481] quote an example where one of these equations (with simple zeros) is Wilkinson's polynomial, see §6.29 on page 80, which is itself a difficult problem.

One of Joab's recent papers [Win11] raises the interesting conjecture of *inexact* coefficients, i.e. different from what they *should* be. Both he and Zhanggong Zeng [Zen05] deploy the notion of a *pejorative manifold* [Kah72] whereby a family of polynomials share a common characteristic, i.e. their multiplicities. However, do different polynomial coefficients denote different polynomial zeros? It is an interesting question which we return to in §7.2.1 on page 96.

Joab's research covers a wide range of topics and he works with two other researchers, Madina Hasan and Xin Lao. Their current research links four recurring topics, shown below.

GCD computation

The best place to start is probably [WLH12], which starts by explaining how the zeros of a polynomial are obtained using the GCD algorithm described at the beginning of this section (starting on page 34), but then demonstates that the operations involved, GCD computations and polynomial divisions, can be ill-posed.

A constrained, perturbed form of the original *inexact polynomial* has an approximate GCD (AGCD). Section 8 of the paper [WLH12, pp. 3487–3488] describes how the *Sylvester resultant matrix*, defined in Equation (2.26) on page 11, is used in this AGCD computation. The perturbated polynomial is constrained so that it is associated with a particular *pejorative manifold*.

Inexact Polynomial

An inexact polynomial is one whose coefficients are, essentially, close approximations to the coefficients of an exact polynomial. These occur naturally when a polynomial is derived from some other calculation which is not exact. If the exact polynomial has multiple zeros, the effect is often that these multiplicities are lost and we have the following [WH13, p. 253].

> "These [*inexact*] polynomials are, therefore, with high probability, coprime, ..."

This results in the *GCD computations* being wrong because the multiplities have disappeared. In [WL11, pp. 1591–1593] Joab (and Xin Lao) describe how polynomial scaling by the geometric mean before computations are carried out gives improved computed results. An analysis of preprocessing with different norms is provided in depth in [WHL12, pp. 246–254].

Pejorative Manifold

Jan Verschelde [Ver11, pp. 1–2] describes the pejorative manifold $\mathcal{M}$ in detail. We will not go into those details here. Joab (and Madina Hasan)[WLH12, p. 3483] re-state their zero finding two-step algorithm in terms of $\mathcal{M}$.

1. Compute the multiplicities of the zeros, i.e. identify the pejorative manifold $\mathcal{M}$ with which $p(z)$ is associated.
2. Compute the values of the zeros, i.e. determine the point on $\mathcal{M}$ that corresponds to $p(z)$.

Step 1 involves the constrained *AGCD* computation mentioned above.

Sylvester Resultant Matrix

This is defined in Equation (2.26) on page 11. Joab (and Madina Hasan) [WH10, p. 3227] point out that the calculation of an approximation of $S(p, q)$ is closely related to the calculation of the $AGCD$ of $p(z)$ and $q(z)$. A later paper by Joab (and Madina) [WH13] describes an improved method for computing the Sylvester resultant matrix.

4.6 Zhonggang Zeng

Zhonggang is a professor of Mathematics in the Department of Mathematics at Northeastern Illinois University.

In [Zen05, pp. 872–873] Zonggang proves the following three lemmas.

Lemma 4.1. *Let $p(z)$ be defined as in Equation (2.3) on page 6, and let p' be its derivative with $g = \gcd(p,\ p')$. Let $S_i(p)$, the **Sylvester discriminant matrix**, be defined as in Equation (2.24) on page 10. For $i = 1,\ 2,\ \ldots,\ n$, let ζ_i be the smallest singular value of $S_i(p)$. Then the following are equivalent.*

(a) $deg(g) = m$;

(b) p has $N = n - m$ distinct zeros;

(c) $\zeta_1,\ \zeta_2,\ \ldots,\ \zeta_{N-1} > 0, \quad \zeta_N = \zeta_{N+1} = \cdots = \zeta_n = 0$;

Lemma 4.2. *Let $p(z)$ be defined as in Equation (2.3) on page 6, and let p' be its derivative with $g = \gcd(p,\ p')$. Let $f(z)$ and $h(z)$ be polynomials satisfying*

$$\begin{aligned} f(z)g(z) &= p(z) \quad , \\ g(z)h(z) &= p'(z) \quad . \end{aligned} \tag{4.38}$$

Then

(a) f and h are co-prime;

(b) the (column) rank of $S_N(p)$ is deficient by one;

(c) the normalised vector $\begin{bmatrix} \mathbf{f} \\ -\mathbf{h} \end{bmatrix}$ is the right singular vector of $S_N(p)$ associated with the smallest (zero) singular value ζ_N;

(d) if $\mathbf{f}$ is known, the coefficient vector $\mathbf{g}$ of $g = \gcd(p,\ p')$ is the solution to the linear system $C_m(f)\mathbf{g} = \mathbf{p}$.

Lemma 4.3. *Let A be a matrix of size $m \times n$ with $m \geq n$ whose smallest two distinct singular values are $\hat{\sigma} > \tilde{\sigma}$. Let $Q \begin{bmatrix} R \\ 0 \end{bmatrix} = A$ be the QR decomposition*

of A, where Q of size $m \times m$ is unitary and R of size $n \times n$ is upper triangular. From any vector $\mathbf{x}_0$ that is not orthogonal to the right singular subspace of A associated with $\tilde{\sigma}$, generate the sequences $\{\sigma_i\}$ and $\{\mathbf{x}_i\}$ by the inverse iteration

$$\left\{ \begin{array}{lcl} \text{Solve} & R^H \mathbf{y}_i = \mathbf{x}_{i-1} & \text{for } \mathbf{y}_i \\ \text{Solve} & R\mathbf{z}_i = \mathbf{y}_i & \text{for } \mathbf{z}_i \\ \text{Calculate} & \mathbf{x}_i = \frac{\mathbf{z}_i}{\|\mathbf{z}_i\|_2} & \sigma_i = \|R\mathbf{x}_i\|_2 \end{array} \right\} \quad i = 1,\ 2,\ \ldots \tag{4.39}$$

Then $\lim_{i\to\infty} \sigma_i = \lim_{i\to\infty} \|A\mathbf{x}_i\|_2 = \tilde{\sigma}$ and $\sigma_i = \|A\mathbf{x}_i\|_2 = \tilde{\sigma} + O(\tau^i)$ where $\tau = \left(\frac{\tilde{\sigma}}{\tilde{\sigma}}\right)^2$.

If $\tilde{\sigma}$ is simple, then $\mathbf{x}_i$ converges to the right singular vector $\tilde{\mathbf{x}}$ of A associated with $\tilde{\sigma}$.

The quadratic system for unknown vectors $\mathbf{f}$, $\mathbf{g}$ and $\mathbf{h}$ is

$$\text{Solve} \begin{bmatrix} g_0 \\ \text{conv}(\mathbf{f},\ \mathbf{g}) \\ \text{conv}(\mathbf{g},\ \mathbf{h}) \end{bmatrix} = \begin{bmatrix} 1 \\ \mathbf{p} \\ \mathbf{p}' \end{bmatrix}, \text{for} \begin{pmatrix} \mathbf{f} \\ \mathbf{g} \\ \mathbf{h} \end{pmatrix} \tag{4.40}$$

where the convolution $\text{conv}(\cdot,\ \cdot)$ is defined by Equation (2.23) on page 10.

Let $m = n - N$ be the degree of $g = \gcd(p,\ p')$, then the Jacobian of the quadratic system given by Equation (4.40) on page 38 is

$$J(\mathbf{f},\ \mathbf{g},\ \mathbf{h}) = \begin{bmatrix} e_1^T & & \\ C_N(g) & C_m(f) & \\ & C_m(h) & C_{N-1}(g) \end{bmatrix}, \tag{4.41}$$

where $e_1 = (1,\ 0,\ \ldots, 0)^T$ and $J(\mathbf{f},\ \mathbf{g},\ \mathbf{h})$ is of full (column) rank.

Let $\tilde{g} = \gcd(p,\ p')$ with $\tilde{f}$ and $\tilde{h}$ satisfying Equation (4.40) on page 38, and let W be a weight matrix. Then there exists $\epsilon > 0$ such that for all $\mathbf{f}_0$, $\mathbf{g}_0$, and $\mathbf{h}_0$ satisfying $\|\mathbf{f}_0 - \tilde{\mathbf{f}}\|_2 < \epsilon$, $\|\mathbf{g}_0 - \tilde{\mathbf{g}}\|_2 < \epsilon$, and $\|\mathbf{h}_0 - \tilde{\mathbf{h}}\|_2 < \epsilon$, the Gauss-Newton iteration, given below, converges quadratically.

$$\begin{bmatrix} \mathbf{f}_{i+1} \\ \mathbf{g}_{i+1} \\ \mathbf{h}_{i+1} \end{bmatrix} = \begin{bmatrix} \mathbf{f}_i \\ \mathbf{g}_i \\ \mathbf{h}_i \end{bmatrix} - J(\mathbf{f}_i,\ \mathbf{g}_i,\ \mathbf{h}_i)^+_W \begin{bmatrix} e_1^T & - & 1 \\ \text{conv}(\mathbf{f}_\text{i},\ \mathbf{g}_\text{i}) & - & \mathbf{p} \\ \text{conv}(\mathbf{g}_\text{i},\ \mathbf{h}_\text{i}) & - & \mathbf{p}' \end{bmatrix}, i = 0,\ 1,\ \ldots\ , \tag{4.42}$$

where $J(\cdot)^+_W = [J(\cdot)^H W^2 J(\cdot)]^{-1} J(\cdot)^H W^2$ is the weighted pseudo-inverse of the

Jacobian $J(\cdot)$ defined in Equation (4.41) on page 38.

Algorithm 4.1. *The following six steps are used to compute* $\gcd(p,\ p')$.

1. *Compute* $m = \deg(p)$ *by inverse iteration to find the smallest singular value* ζ_k *of* $S_k(p)$ *and the corresponding right singular vector.*

2. *Calculate the QR decomposition of the* $(n+1) \times 3$ *matrix* $S_1(p) = Q_1 R_1$.

3. *For* $i = 1,\ 2,\ \cdots,\ n-1$ *do*

   (a) *Use the inverse iteration, Equation (4.39) on page 38, to find the smallest singular value* ζ_i *of* $S_i(p)$ *and the corresponding* $\mathbf{y}_i$.

   (b) *If* $\zeta_i \leq \theta \|\mathbf{p}\|_2$, *then* $N = i$, $m = n - N$, *extract* $\boldsymbol{f}$ *and* $\boldsymbol{h}$ *from* $\mathbf{y}_i$, *and exit*
   Else update $S_i(p)$ *to* $S_{i+1}(p) = Q_{i+1} R_{i+1}$
   End if

   End do Note that Zhonggang Zeng states that θ *is a* ***zero singular value threshold****; consult [Zen05, p. 895] for further details.*

4. *Set up the quadratic GCD system, Equation (4.40) on page 38, in accordance with the degree* m.

5. *In step 1, when the singular value* ζ_i *is calculated, the associated singular vector* $\mathbf{y}_i$ *consists of* $\mathbf{f}_0$ *and* $\mathbf{h}_0$, *which are approximations to* $\mathbf{f}$ *and* $\mathbf{h}$ *in Equation (4.38) on page 37.*

   The linear system

   $$C_m \mathbf{f}_0 \mathbf{g}_0 = \mathbf{p} \tag{4.43}$$

   is solved for a least squares solution $\mathbf{g}_0$ *that minimises* $\|\mathrm{conv}(\mathbf{f}_0,\ \mathbf{g}_0) - \mathbf{p}\|$.

6. *Use the Gauss-Newton iteration, Equation (4.42) on page 38, to refine the GCD triplet* $(f,\ g,\ h)$. *This iteration is expected to reduce the residual*

   $$\begin{aligned} &\left\| \begin{pmatrix} \mathrm{conv}(\mathbf{f}_i,\ \mathbf{g}_i) \\ \mathrm{conv}(\mathbf{g}_i,\ \mathbf{h}_i) \end{pmatrix} - \begin{pmatrix} \mathbf{p} \\ \mathbf{p}' \end{pmatrix} \right\|_W \\ = &\left\| W \begin{pmatrix} \mathrm{conv}(\mathbf{f}_i,\ \mathbf{g}_i) & - & \mathbf{p} \\ \mathrm{conv}(\mathbf{g}_i,\ \mathbf{h}_i) & - & \mathbf{p}' \end{pmatrix} \right\|_2 \end{aligned} \tag{4.44}$$

   at each step until it is numerically irreducible. The diagonal weight matrix W *is used to scale the GCD system, Equation (4.40) on page 38, so that the entries of* $W \begin{bmatrix} \mathbf{p} \\ \mathbf{p}' \end{bmatrix}$ *are of similar magnitude. Each step of the Gauss-*

Newton iteration requires solving an over-determined linear system

$$[WJ(\mathbf{f}_i,\ \mathbf{g}_i,\ \mathbf{h}_i)]\,\mathbf{z} = W \begin{bmatrix} e_1^T\mathbf{u}_i & - & 1 \\ \text{conv}(\mathbf{f}_i,\ \mathbf{g}_i) & - & \mathbf{p} \\ \text{conv}(\mathbf{g}_i,\ \mathbf{h}_i) & - & \mathbf{p}' \end{bmatrix} \tag{4.45}$$

for its least squares solution $\mathbf{z}$.

4.6.1 Factorisation

The following process can be used to factor the polynomial $p(z)$.

Algorithm 4.2. *This algorithm consists of a single loop containing three steps. Once again,* $g_0(z) = p(z)$.

For $i = 1,\ 2,\ \ldots,$ *while* $\deg(g_{i-1}(z)) > 0$ *do*

1. *Calculate* $g_i(z) = \gcd(g_{i-1}(z),\ g'_{i-1}(z))$.
2. *Calculate* $f_i(z) = \dfrac{g_{i-1}(z)}{g_i(z)}$.
3. *Calculate the (simple) zeros of* $f_i(z)$.

End do

4.6.2 Computing the Multiplicities

The process above generates a sequence of square-free polynomials

$$f_1(z),\ f_2(z),\ \ldots,\ f_M(z) \tag{4.46}$$

of degrees $d_1 \geq d_2 \cdots \geq d_M$, respectively. So $N = d_1 = \deg(f_1(z))$. Then the multiplicities $\{m_i\}$ are computed as follows.

$$m_i = \max\{j \mid d_j \geq (d_1 + 1) - i\} \quad ,\quad i = 1,\ 2,\ \ldots,\ N \quad . \tag{4.47}$$

Formal justification for this approach can be found in [Zen05, p. 894]. Zhonggang's latest work on inexact polynomials, which is very similar to the approach taken by Joab Winkler, see §4.5 starting on page 34, is found in [Zen09].

He has also initiated a project with another mathematician, Tien-Yien Li from Michigan State University, currently entitled NAClab 2.0 – a Numerical Algebraic Computing Toolbox for Matlab [Zen12] which is, to quote from their web site.

> "A robust software package for accurate numerical solution of algebraic problems assuming the data are perturbed, along with programming tools for further research and development."

It is interesting that we are seeing more research going into this area of perturbed data values at this time, e.g. see towards the end of Joab Winkler's entry, which starts at §4.5 on page 34.

Chapter 5

Our Research in This Area

This chapter presents the algorithms and results of our research into locating the zeros of polynomials.

The research culminating in this thesis was originally carried out in collaboration with Professor George Loizou. George is currently Professor Emeritus of the Mathematics of Computation in the Department of Computer Science and Information Systems, Birkbeck, University of London.

During the search stage we have not applied any scaling to the original format of the polynomial. So far, we have not found it necessary to apply any scaling.

During the iterative stage the polynomial is scaled into monic form, i.e. $a_n = 1$ in Equation (2.3) on page 6. In adddition, we should emphasise that we use *single-point* IFs, i.e. we evaluate the polynomial and its derivatives at just at one point z_ν. This is in contrast to *multipoint* IFs which evaluate the polynomial and its derivatives at intermediate points in order to derive higher-order IFs.

We present classes of IFs as part of the iterative stage. Each class contains representatives from second-order through to fifth-order. Using Matlab, there is very little difficulty in automatically generating IFs of higher-orders, but we must ask what would be gained in doing so. The actual IFs are already complex, in terms of their structure. Higher-order IFs mean evaluating more polynomial derivatives and programming more complex code, but to what gain? Possibly quicker convergence (in terms of iterations) at best? No, we believe that second-order to fifth-order IFs are sufficient to illustrate each class.

We also do not present all of those IFs that make minor improvements to the order of convergence (of other researchers' IFs). For example, George Loizou's paper [Loi83], and Richard King's paper [Kin83] (which improves Newton's

method to an order of convergence of $(1 + \sqrt{2})$), which is similar to a paper by Peter Kravanja and Ann Hergemans [KH99].

5.1 Computational Methodology

REMARK 3. *Section added. Winkler point 2.*

When we originally embarked on our research in this area (the late 1960s) the only practical programming language to use was Fortran, for two reasons.

- It was the language of choice for numerical analysis.
- Multiple precision libraries were becoming available.

Both of us were fluent in Algol 60 [B$^+$63], an obvious choice, but it was lacking in support libraries, apart from the nascent NAG library [FB77], which was being funded in the UK.

There was then a break in our work, ...

When we returned to this work (the early 2000s), our programming language of choice was C, emanating from our use of the UNIX operating system for teaching and research. By then, the GNU project was well under way, and one of its most successful products was the GNU Multiple Precision Arithmetic Library (GMP) [Gra11].

Combining the two, together with C's powerful macro facilities, meant that our algorithms could be expressed in a vector-oriented high-level notation that made programming easier and natural.

Our decision to use multiple precision arithmetic to overcome any problems with rounding errors in our computations meant that we did not consider alternatives, such as Matlab [Mat13] and Mathematica [Ber56], which did not have multiple precision libraries at that time, for our implementation. As will be seen in subsequent sections, this decision was the correct one.

5.2 Search Stage

REMARK 4. *Initial three paragraphs inserted. Lai point 1. Winkler point 4.*

This stage is concerned with locating regions within the complex plane known to contain zeros of a given polynomial. We start by locating a circular region containing all the zeros. We then find the smallest square region containing this

circular region. We then divide this square into four equal squares (our default), test each square to determine whether it contains zeros or not, and retain only those squares containing zeros.

This sub-division process continues until the squares are small enough for their centres to be taken as approximations to the true zeros.

Squares are used because they completely cover the complex plane without overlap. Unfortunately, our test for zeros only works on circles; these can overlap and cause zero(s) in one square to also be counted within an adjacent square. This problem is alleviated by a correction mechanism, described in §5.2.2 starting on page 47, that eradicates those zeros counted twice.

In our paper [FL75] we were concerned only with deriving some new families of IFs. For our later paper [FL77, pp. 431–433] we wanted to automate the complete algorithm by computing initial approximations to the zeros rather than giving them as arguments to the various IFs. This old approach was always doubtful as these approximations could be *modified a priori* in order to get a favourable situation for the IFs to converge.

Our new approach was to first locate an *inclusion region* containing all the zeros of the polynomial. We chose the circular region about the origin derived by Dirk Dekker [Dek68] containing all the zeros. This is given by Equation (3.1) on page 14.

This circle is covered by a number of smaller circles, each with the same area. One possible covering is given in Figure 5.1. Another possible covering is given

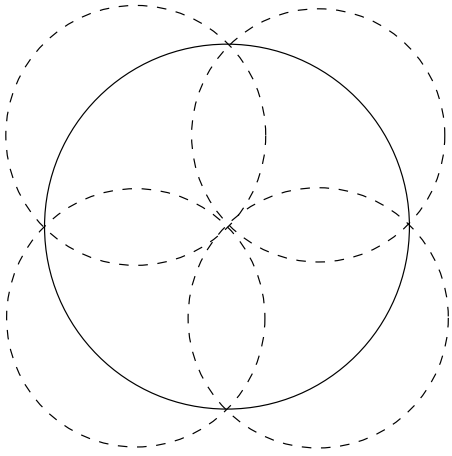

Figure 5.1: Covering with Four Circles

in Figure 5.2 (there are, of course, others). The following algorithm is then applied.

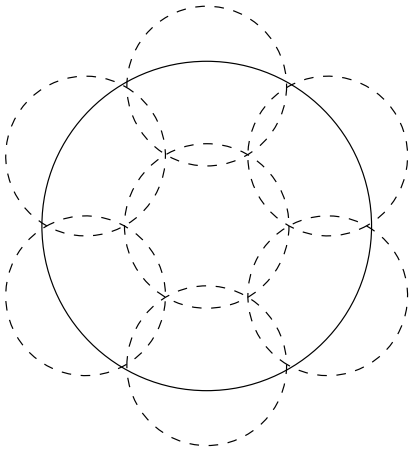

Figure 5.2: Covering with Seven Circles

Algorithm 5.1. *The algorithm consists of the following three steps.*

1. *Each circle is tested to see if it contains at least one zero of $p(z)$. The test used was that stated by Dick Lehmer, namely Theorem 3.3 on page 20. The mapping of the circles to the unit circle was that given by Professor Stewart, namely Algorithm 3.2 on page 24.*

   Retain only those circles that test positively.

2. *Cover these circles using the same type of covering as before.*

3. *Re-apply the previous two steps until the circles containing the zeros have separated.*

As the radii of the circles are monotonically decreasing, their centres are successively improving approximations to the zeros.

The multiplicities of these approximations can be estimated by using Jean-Louis Lagouanelle's bounding formula, given by Equation (3.5) on page 18. Other limiting formulae could be used, for example those described by Dirk Dekker [Dek68, pp. 193–194] and Joseph Traub [Tra64, pp. 154–157].

The search procedure is continued until

$$\sum_i m_i = n \quad , \tag{5.1}$$

when approximations are known to all the factors of the polynomial $p(z)$.

5.2.1 Search Efficiency

A major problem with using circles as approximations to zeros is the amount of overlap between adjacent circles. Any zero in this overlap will be counted as being in both circles. This situation can continue until sufficient separation is obtained.

Consider the scenario when using a covering of four circles as illustrated in Figure 5.1 on page 44. Given an initial circle of radius R (solid line) there is a *total* overlap (dashed lines) of

$$R^2(\pi - 2) \quad , \tag{5.2}$$

which is approximately 36% of the initial circle's area.

The situation is slightly better when using a covering of seven circles as illustrated in Figure 5.2 on page 45. Given an initial circle of radius R (solid line) there is a total overlap (dashed lines) of

$$R^2(\pi - \frac{3\sqrt{3}}{2}) \quad , \tag{5.3}$$

which is approximately 17% of the initial circle's area.

For our paper [FL85b] we decided to use square regions instead of circles because this decreases the amount of overlap.

Our first inclusion region is the square covering the circle given by Dirk Dekker's formula, as illustrated by Figure 3.1 on page 15.

Figure 5.3 illustrates such a square covered by four smaller squares and nine smaller squares. A circular inclusion test is applied to each smaller square, so overlap still occurs.

Given an initial square of radius (semi-diagonal) R there is a total overlap of

$$\frac{N-1}{N} R^2(\pi - 2) \tag{5.4}$$

when the initial square is covered by N^2 smaller squares. This is approximately 29% of the initial square's area when $N = 2$, rising asymptotically to 57% as N increases towards infinity. From this, it is apparent a covering of four squares is preferable, all other things being equal.

For our paper [FL85b] we also utilise Morris Marden's Theorem 3.6 on page 21, which returns the *number* of zeros in the unit circle. For us, this yields the number of zeros in a collection of squares. This number could be larger than n,

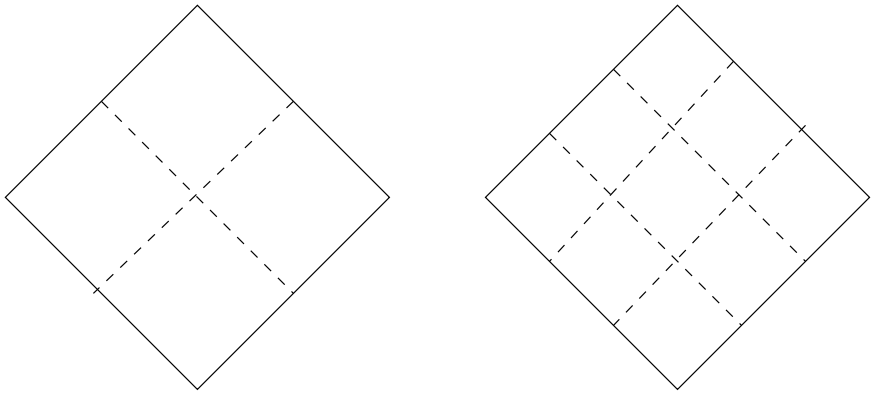

Figure 5.3: Covering with Squares

the degree of $p(z)$, because of the overlaps introduced by the circular inclusion test.

However, when used in conjunction with Jean-Louis Lagouanelle's estimates of the multiplicities, Equation (3.5) on page 18, we have better control over when to terminate the search stage, i.e. only when the two counts agree for each approximation, given by the centre of the square.

This is a useful safety net. Occasionally, the sum of Morris Marden's estimations equals the degree of the polynomial whereas Jean-Louis Lagouanelle's estimations are more conservative.

5.2.2 Search Consolidation

Although the overlap when using squares is less than with circles, it can still occur. This is overcome using a *consolidation* stage after a certain number of iterations. Consider Figure 5.4. Within a general covering of squares (dotted lines) two adjacent squares (solid lines) test positive for containing a zero α_1 because it lies within the overlap of the circular inclusion test.

We now cover our collection of small squares with a collection of larger ones (dashed lines); these larger squares subsume all small squares, containing zeros, that they contain. Note that in Figure 5.4 this includes a second square nearby containing another zero α_2.

The search stage then continues using the larger squares. This operation shifts the centres of the squares tested, and therefore the overlaps, and has proved very successful in practice, see Chapter 7 starting on page 87.

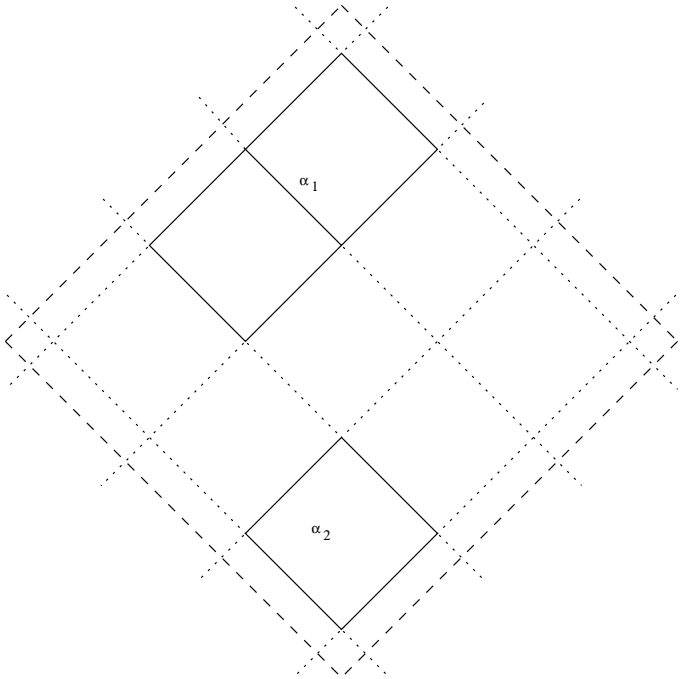

Figure 5.4: Consolidating Squares

5.2.3 Search Convergence

REMARK 5. *This new section added. George's idea.*

In theory it is possible to continue this search stage until the squares are so small that their centres can be taken as accurate approximations to the true zeros.

However, the computional workload is substantial and would result in obtaining our desired results slower than necessary. For this reason we stop this search stage when we consider the centres of our squares to be resonable approximations to the true zeros and switch to the faster IFs, described in the next section, for rapid convergence.

5.3 Simple Zeros

Our paper [FL75] presented a family of IFs for improving the approximations to the zeros of a polynomial with only simple zeros.

This family, or class, of IFs is derived from the Taylor series expansion of $p(\alpha_\nu)$

about z_ν, namely

$$p(\alpha_\nu) = 0 = p(z_\nu) + \sum_{i=1}^{\rho-1} \frac{(-1)^i}{i!} p^{(i)}(z_\nu)\epsilon_\nu^i + O(\epsilon_\nu^\rho) \quad . \tag{5.5}$$

Using the auxiliary definitions in Equation (2.6) on page 6 and Equation (2.7) on page 6, we can re-write Equation (5.5) as

$$u(z_\nu) = \sum_{i=1}^{\rho-1} (-1)^{i-1} A_i(z_\nu)\epsilon_\nu^i + O(\epsilon_\nu^\rho) \quad . \tag{5.6}$$

Omitting the terms of $O(\epsilon_\nu^\rho)$ in Equation (5.6) leaves a polynomial of degree $\rho - 1$ in ϵ_ν. Joseph Traub [Tra64, pp. 61–62] points out that this could be used as a non-linear equation for low values of ρ. However, Equation (5.6) can be rearranged to obtain our family of IFs. They can be generated using the following equations. We start with $\rho = 2$, namely

$$\lambda_2(z_\nu) = u(z_\nu) \quad . \tag{5.7}$$

Now, as ρ increases, we substitute ϵ_ν^i in Equation (5.6) with the appropriate value for powers of $\lambda_i(z_\nu)$ as follows

$$\begin{aligned}
\lambda_2(z_\nu) &= u(z_\nu) \quad , \\
\lambda_3(z_\nu) &= u(z_\nu) + A_2(z_\nu)\lambda_2^2(z_\nu) \quad , \\
\lambda_4(z_\nu) &= u(z_\nu) + A_2(z_\nu)\lambda_3^2(z_\nu) - A_3(z_\nu)\lambda_2^3(z_\nu) \quad , \\
\lambda_5(z_\nu) &= u(z_\nu) + A_2(z_\nu)\lambda_4^2(z_\nu) - A_3(z_\nu)\lambda_3^3(z_\nu) + A_4(z_\nu)\lambda_2^4(z_\nu) \quad , \\
\lambda_6(z_\nu) &= u(z_\nu) + A_2(z_\nu)\lambda_5^2(z_\nu) - A_3(z_\nu)\lambda_4^3(z_\nu) + A_4(z_\nu)\lambda_3^4(z_\nu) \\
&\quad - A_5(z_\nu)\lambda_2^5(z_\nu) \quad ,
\end{aligned} \tag{5.8}$$

etc. Then the IF

$$R_\rho(z_\nu) = z_\nu - \lambda_\rho(z_\nu) \tag{5.9}$$

is of order ρ. Simplifying the equation for $\lambda_\rho(z_\nu)$ into another equation involving $A_2(z_\nu), A_3(z_\nu) \ldots A_{\rho-1}(z_\nu)$ and the powers of $u(z_\nu)$ is time-consuming and error-prone. Fortunately, Joseph Traub provides a recurrence relation for the IFs in [Tra64, p. 87] which simplifies the calculations, namely (using our notation)

$$R_{\rho+1}(z_\nu) = R_\rho(z_\nu) - \frac{u(z_\nu)}{\rho} R'_\rho(z_\nu) \quad . \tag{5.10}$$

Alternatively, this relation can be expressed in terms of our $\lambda_2(z_\nu)$, $\lambda_3(z_\nu)$, ...

as follows

$$\lambda_{\rho+1}(z_\nu) = \lambda_\rho(z_\nu) + \frac{u(z_\nu)}{\rho}[1 - \lambda'_\rho(z_\nu)] \quad . \tag{5.11}$$

Finally, we can now express the various $\lambda_i(z_\nu)$ as polynomials in $u(z_\nu)$. Combining the entries in Equation (5.8) on page 49 yields the following

$$\begin{aligned}
\lambda_2(z_\nu) &= u(z_\nu) \quad , \\
\lambda_3(z_\nu) &= \lambda_2(z_\nu) + A_2(z_\nu)u^2(z_\nu) \quad , \\
\lambda_4(z_\nu) &= \lambda_3(z_\nu) + [2A_2^2(z_\nu) - A_3(z_\nu)]u^3(z_\nu) \quad , \\
\lambda_5(z_\nu) &= \lambda_4(z_\nu) + [5A_2^3(z_\nu) - 5A_2(z_\nu)A_3(z_\nu) + A_4(z_\nu)]u^4(z_\nu) \quad , \\
\lambda_6(z_\nu) &= \lambda_5(z_\nu) + [14A_2^4(z_\nu) - 21A_2^3(z_\nu)A_3(z_\nu) + 6A_2(z_\nu)A_4(z_\nu) \\
&\qquad + 3A_3^2(z_\nu)]u^5(z_\nu) \quad ,
\end{aligned} \tag{5.12}$$

etc. Higher-order powers of $u(z_\nu)$ than those required for the IFs have been removed.

These equations have all been verified using the symbolic manipulation module of Matlab [Mat12].

Note that the IFs presented hereafter are in their *polynomial* form rather than the possibly more familiar *rational* form. This polynomial form makes it easier to see how the IFs grow in complexity as the order of convergence increases. It also facilitates our work in computing the asymptotic error constants in Appendix A.

For completeness, the rational forms of these IFs can be found in Appendix B starting on page 120.

5.3.1 A Class of One-point IFs for Simple Zeros

The first five IFs, $R_\rho(z_\nu)$, are given below.

Isaac Newton's second-order IF [New36]

$$\hat{z}_\nu = z_\nu - u(z_\nu) \quad . \tag{5.13}$$

The rational form of this IF is Equation (B.1) on page 120.

Edmond Halley's third-order IF [Hal94]

$$\hat{z}_\nu = z_\nu - u(z_\nu) - A_2(z_\nu)u^2(z_\nu) \quad . \tag{5.14}$$

The rational form of this IF is Equation (B.2) on page 121.

I Kiss' fourth-order IF [Kis54, p. 68]

$$\hat{z}_\nu = z_\nu - u(z_\nu) - A_2(z_\nu)u^2(z_\nu) - [2A_2^2(z_\nu) - A_3(z_\nu)]u^3(z_\nu) \quad . \tag{5.15}$$

The rational form of this IF is Equation (B.3) on page 121.

I Kiss' fifth-order [Kis54, p. 68]

$$\begin{aligned}\hat{z}_\nu = z_\nu &- u(z_\nu) - A_2(z_\nu)u^2(z_\nu) \\ &- [2A_2^2(z_\nu) - A_3(z_\nu)]u^3(z_\nu) \\ &- [5A_2^3(z_\nu) - 5A_2(z_\nu)A_3(z_\nu) + A_4(z_\nu)]u^4(z_\nu)\end{aligned} \quad . \tag{5.16}$$

Joseph Traub [Tra64, p. 84] states the polynomial form in Equation (5.16) above, while Kiss derived the original rational form in Equation (B.4) on page 121.

Mick Farmer and George Loizou's sixth-order IF

$$\begin{aligned}\hat{z}_\nu = z_\nu &- u(z_\nu) - A_2(z_\nu)u^2(z_\nu) \\ &- [2A_2^2(z_\nu) - A_3(z_\nu)]u^3(z_\nu) \\ &- [5A_2^3(z_\nu) - 5A_2(z_\nu)A_3(z_\nu) + A_4(z_\nu)]u^4(z_\nu) \\ &- [14A_2^4(z_\nu) - 21A_2^2(z_\nu)A_3(z_\nu) + 6A_2(z_\nu)A_4(z_\nu) \\ &\qquad + 3A_3^2(z_\nu) - A_5(z_\nu)]u^5(z_\nu)\end{aligned} \quad . \tag{5.17}$$

We have found no previous derivation of this IF, although Joseph Traub [Tra64, p. 84] does provide the coefficient of the last term in one of his tables.

The IFs defined by Equation (5.13) through Equation (5.16) have been verified using a Matlab program, see §C.6.2 on page 200.

5.3.2 A Class of Simultaneous IFs for Simple Zeros

This class of IFs is generated by replacing the derivative of highest order, $A_{\rho-1}(z_\nu)$ in R_ρ, in one of our earlier one-point IFs (see §5.3.1 starting on page 50) with a first-order approximation involving $T_{\rho-2}(z_\nu)$, as defined in Equation (2.13) on page 7.

Using $p(z)$, as defined in Equation (2.4) on page 6, and $S_k(z)$, as defined in

Equation (2.11) on page 7, we have

$$\begin{aligned}\frac{p'(z_\nu)}{p(z_\nu)} &= \frac{\sum_i \prod_{j\neq i}(z_\nu - \alpha_j)}{\prod_j (z_\nu - \alpha_j)} \quad , \\ &= \sum_i \frac{1}{z_\nu - \alpha_i} \quad , \\ &= \frac{1}{\epsilon_\nu} + S_1(z_\nu) \quad .\end{aligned} \tag{5.18}$$

Following the same approach as with Equation (5.18), we have

$$\begin{aligned}\frac{p''(z_\nu)}{p(z_\nu)} &= \frac{\sum_i \sum_{j\neq i} \prod_{k\neq i,j} (z_\nu - \alpha_k)}{\prod_k (z_\nu - \alpha_k)} \quad , \\ &= \sum_i \sum_{j\neq i} \frac{1}{(z_\nu - \alpha_i)(z_\nu - \alpha_j)} \quad , \\ &= \sum_i \frac{1}{z_\nu - \alpha_i} \sum_{j\neq i} \frac{1}{z_\nu - \alpha_j} \quad , \\ &= \sum_i \frac{1}{z_\nu - \alpha_i} \left[\frac{1}{\epsilon_\nu} + S_1(z_\nu) - \frac{1}{z_\nu - \alpha_i} \right] \quad , \\ &= \left[\frac{1}{\epsilon_\nu} + S_1(z_\nu) \right] \sum_i \frac{1}{z_\nu - \alpha_i} - \sum_i \frac{1}{(z_\nu - \alpha_i)^2} \quad , \\ &= \left[\frac{1}{\epsilon_\nu} + S_1(z_\nu) \right]^2 - \frac{1}{\epsilon_\nu^2} - S_2(z_\nu) \quad , \\ &= S_1^2(z_\nu) - S_2(z_\nu) + \frac{2}{\epsilon_\nu} S_1(z_\nu) \quad .\end{aligned} \tag{5.19}$$

Finally, dividing Equation (5.19) by Equation (5.18) yields

$$\begin{aligned}A_2(z_\nu) &= \frac{p''(z_\nu)}{2! p'(z_\nu)} \quad , \\ &= \frac{2S_1(z_\nu) + [S_1^2(z_\nu) - S_2(z_\nu)]\epsilon_\nu}{2[1 + S_1(z_\nu)\epsilon_\nu]} \quad , \\ &= S_1(z_\nu) + O(\epsilon_\nu) \quad .\end{aligned} \tag{5.20}$$

Successively differentiating the middle line of Equation (5.20) yields the set of

equations

$$\begin{aligned} A_2(z_\nu) &= S_1(z_\nu) + O(\epsilon_\nu) \quad , \\ 2A_3(z_\nu) &= A_2^2(z_\nu) - S_2(z_\nu) + O(\epsilon_\nu) \quad , \\ 3A_4(z_\nu) &= -A_2^3(z_\nu) + 3A_2(z_\nu)A_3(z_\nu) + S_3(z_\nu) + O(\epsilon_\nu) \quad . \end{aligned} \tag{5.21}$$

Given that

$$S_k(z_\nu) = T_k(z_\nu) + O(\epsilon_\nu) \quad , \tag{5.22}$$

we finally arrive at the following set of equations used in deriving this class of IFs.

$$\begin{aligned} A_2(z_\nu) &= T_1(z_\nu) + O(\epsilon_\nu) \quad , \\ 2A_3(z_\nu) &= A_2^2(z_\nu) - T_2(z_\nu) + O(\epsilon_\nu) \quad , \\ 3A_4(z_\nu) &= -A_2^3(z_\nu) + 3A_2(z_\nu)A_3(z_\nu) + T_3(z_\nu) + O(\epsilon_\nu) \quad . \end{aligned} \tag{5.23}$$

Replacing the derivative of highest order, i.e. $A_{\rho-1}(z_\nu)$, in R_ρ by a suitable approximation from the above equations retains the order of convergence while one less derivative needs computing at every step. The first few such IFs are given below.

Second-order IF

There is no polynomial form for this second-order IF. However, its well-known rational form, derived by Kiril Dochev and P Byrnev, is Equation (B.5) on page 121.

Mick Farmer and George Loizou's third-order IF

Substituting for $A_2(z_\nu)$ in Equation (5.14) on page 50 yields the following third-order IF.

$$\hat{z}_\nu = z_\nu - u(z_\nu) - T_1(z_\nu)u^2(z_\nu) \quad . \tag{5.24}$$

The rational form of this IF, derived by Louis Ehrlich, is Equation (B.6) on page 121.

Mick Farmer and George Loizou's fourth-order IF

Substituting for $A_3(z_\nu)$ in Equation (5.15) on page 51 yields the following fourth-order IF.

$$\hat{z}_\nu = z_\nu - u(z_\nu) - A_2(z_\nu)u^2(z_\nu) - \frac{1}{2}[3A_2^2(z_\nu) + T_2(z_\nu)]u^3(z_\nu) \quad . \tag{5.25}$$

We previously derived this IF in its rational form. This is Equation (B.7) on page 122.

Mick Farmer and George Loizou's fifth-order IF

Substituting for $A_4(z_\nu)$ in Equation (5.16) on page 51 yields the following fifth-order IF.

$$\begin{aligned}\hat{z}_\nu = z_\nu &- u(z_\nu) - A_2(z_\nu)u^2(z_\nu) \\ &- [2A_2^2(z_\nu) - A_3(z_\nu)]u^3(z_\nu) \\ &- \frac{1}{3}[14A_2^3(z_\nu) + 12A_2(z_\nu)A_3(z_\nu) + T_3(z_\nu)]u^4(z_\nu)\end{aligned} \quad . \tag{5.26}$$

We previously derived this IF in its rational form. This is Equation (B.8) on page 122.

The IFs defined by Equation (5.24) through Equation (5.26) have been verified using a Matlab program, see §C.6.2 on page 200.

In addition, these IFs generate approximations to the zeros α_ν in *parallel*, i.e. Jacobi-like. Once better approximations have been computed, it is advantageous to compute the remaining approximations in *serial*, i.e Gauss-Seidel-like, using the newer approximations as soon as they become available. For example, the third-order IF given in Equation (5.24) on page 53 becomes

$$\hat{z}_\nu = z_\nu - u(z_\nu) - \left[\sum_{i<\nu}(z_\nu - \hat{z}_i) + \sum_{i>\nu}(z_\nu - z_i)\right] u^2(z_\nu) \quad . \tag{5.27}$$

This is a classic example of what is known as in the literature as *series acceleration*, i.e. the new sequence converges faster than the original sequence at no extra cost. The formal term for the study of this technique is called **R-order convergence**. This has already been mentioned in the work of Miodrag Petković, starting with Equation (4.23) on page 32, and will be alluded to again in a section of our chapter on further work, namely §9.4 starting on page 105.

5.3.3 A Class of Variable-order IFs for Simple Zeros

The idea behind this class is very simple. Instead of using the factor $z_\nu - z_i$ in the IFs derived above, we use $z_\nu - \hat{z}_i$ where $\hat{z}_i$ is an improved *and updated* approximation to z_i using an application of Newton's second-order IF given by Equation (5.13) on page 50.

Other authors, principally Abdel Anourein [Ano77, pp. 244–245] and Miodrag Petković [Pet89, pp.83–84], have also used a Newton correction to improve convergence, but both *do not* update the value of z_i at the same time. This is obviously extremely wasteful, as we shall show.

So, for an IF with order of convergence ρ, while computing $\hat{z}_\nu$ each of the other approximations is improved with an order of convergence of two. Therefore, the overall order of convergence increases with the degree of the polynomial. The cost is computing the value of $u(z_i)$ multiple times.

Looked at in simple terms, assuming we start with initial approximations of $O(\epsilon)$, the new order of convergence is roughly $2^{(n-1)}\rho$ where n is the degree of the polynomial.

Based on Kiril Dochev and P Byrnev's second-order IF, Equation (B.5)

We have not exploited this IF as one of the reasons for using Equation (B.5) on page 121 is not having to compute $p'(z)$.

Mick Farmer and George Loizou's variable-order IF

This is based on Ehrlich's third-order IF, Equation (5.24) on page 53. Instead of $T_1(z_\nu)$ we use the formula $\displaystyle\sum_{i\neq\nu} \frac{1}{z_\nu - [\hat{z}_i = z_i - u(z_i)]}$ to yield the IF given by

$$\hat{z}_\nu = z_\nu - u(z_\nu) - \left[\sum_{i\neq\nu} \frac{1}{z_\nu - [\hat{z}_i = z_i - u(z_i)]}\right] u^2(z_\nu) \quad . \tag{5.28}$$

We accept that purists would argue that this is not strictly a simple one-point IF. Given that the polynomial is evaluated at different, and *updated*, values of z_i at a lower level means it is strictly an example of a Multipoint IF, see §5.6.2 starting on page 64.

This replacement can obviously be taken to higher levels, but we have not done so in this thesis. However, see §9.2, starting on page 104, for details of our plans.

5.4 Multiple Zeros

In [FL75] we derived a class of IFs for improving the zeros of a polynomial with only simple zeros. In a later paper [FL77] we extended those results to multiple zeros in the context of a *globally convergent* algorithm.

Algorithm 5.2. *This algorithm consists of the following three steps.*

1. *Find an inclusion region of the complex plane containing all the zeros of the polynomial. This is Equation (3.1) on page 14.*

2. *Apply a slowly convergent search algorithm to obtain initial approximations to the zeros and to compute their multiplicities. This is described in Section §5.2 starting on page 43.*

3. *Improve these approximations with a rapidly convergent IF to any required accuracy. This is described below.*

Let $p(z)$ be defined as in Equation (2.3) on page 6. Then the function $P(z_\nu)$ defined by

$$P(z_\nu) = p^{\frac{1}{m_\nu}}(z_\nu) \tag{5.29}$$

has a simple zero corresponding to an m_ν-tuple zero of $p(z)$. The following definitions will be used subsequently.

$$U(z_\nu) = \frac{P(z_\nu)}{P'(z_\nu)} = m_\nu u(z_\nu) \quad , \tag{5.30}$$

$$B_i(z_\nu) = \frac{P^{(i)}(z_\nu)}{i!P'(z_\nu)} \quad . \tag{5.31}$$

Once again, following our approach in [FL75], we can use the Taylor series expansion of $P(\alpha_\nu)$ about z_ν, since $P(z_\nu)$ has a simple zero α_ν, to obtain, after dividing both sides of the series by $P'(z_\nu)$,

$$U(z_\nu) = \sum_{i=1}^{\rho-1}(-1)^{i-1}B_i(z_\nu)\epsilon_\nu^i + O(\epsilon_\nu^\rho) \quad . \tag{5.32}$$

Following the same logic as applied in §5.3, starting on page 48 for simple zeros, Equation (5.30) on page 56 can be rearranged to obtain a different family of IFs.

Now, as ρ increases, substitute ϵ_ν^i in Equation (5.30) on page 56 with the appropriate value for powers of $\Lambda_i(z_\nu)$ as follows

$$\begin{aligned}
\Lambda_2(z_\nu) &= U(z_\nu) \quad , \\
\Lambda_3(z_\nu) &= U(z_\nu) + B_2(z_\nu)\Lambda_2^2(z_\nu) \quad , \\
\Lambda_4(z_\nu) &= U(z_\nu) + B_2(z_\nu)\Lambda_3^2(z_\nu) - B_3(z_\nu)\Lambda_2^3(z_\nu) \quad , \\
\Lambda_5(z_\nu) &= U(z_\nu) + B_2(z_\nu)\Lambda_4^2(z_\nu) - B_3(z_\nu)\Lambda_3^3(z_\nu) + B_4(z_\nu)\Lambda_2^4(z_\nu) \quad , \\
\Lambda_6(z_\nu) &= U(z_\nu) + B_2(z_\nu)\Lambda_5^2(z_\nu) - B_3(z_\nu)\Lambda_4^3(z_\nu) + B_4(z_\nu)\Lambda_3^4(z_\nu) \\
&\qquad - B_5(z_\nu)\Lambda_2^5(z_\nu) \quad ,
\end{aligned} \tag{5.33}$$

etc. Once again, higher-order powers of $U(z)$ than those required for the IFs are removed. Then the IF

$$R_\rho = z_\nu - \Lambda_\rho(z_\nu) \tag{5.34}$$

is of order ρ. As mentioned in [FL77, pp. 428–429], some members of this class of IFs are well known when only simple zeros are present, see §5.3, starting on page 48, for these.

Equation (5.13) on page 50 through Equation (5.16) on page 51 are identical except that $U(z_\nu)$ replaces $u(z)$ and $B_i(z_\nu)$ replaces $A_i(z)$. In order to express these IFs in terms of $u(z_\nu)$ and $A_i(z_\nu)$, we successively differentiate Equation (5.29) on page 56 to obtain the following set of equations.

$$
\begin{aligned}
B_2(z_\nu) &= -\frac{m_\nu - 1}{2!\, m_\nu u(z_\nu)} + A_2(z_\nu) \quad , \\
B_3(z_\nu) &= \frac{(m_\nu - 1)(2m_\nu - 1)}{3! m_\nu^2 u^2(z_\nu)} - \frac{m_\nu - 1}{m_\nu u(z_\nu)} A_2(z_\nu) + A_3(z_\nu) \quad , \\
B_4(z_\nu) &= -\frac{(m_\nu - 1)(2m_\nu - 1)(3m_\nu - 1)}{4! m_\nu^3 u^3(z_\nu)} \\
&\quad + \frac{(m_\nu - 1)(2m_\nu - 1)}{2! m_\nu^2 u^2(z_\nu)} A_2(z_\nu) \\
&\quad - \frac{(m_\nu - 1)}{m_\nu u(z_\nu)} \left[\frac{A_2^2(z_\nu)}{2!} + A_3(z_\nu) \right] + A_4(z_\nu) \quad , \\
B_5(z_\nu) &= \frac{(m_\nu - 1)(2m_\nu - 1)(3m_\nu - 1)(4m_\nu - 1)}{5! m_\nu^4 u^4(z_\nu)} \\
&\quad - \frac{(m_\nu - 1)(2m_\nu - 1)(3m_\nu - 1)}{3! m_\nu^3 u^3(z_\nu)} A_2(z_\nu) \\
&\quad + \frac{(m_\nu - 1)(2m_\nu - 1)}{2! m_\nu^2 u^2(z_\nu)} \left[A_2^2(z_\nu) + A_3(z_\nu) \right] \\
&\quad - \frac{(m_\nu - 1)}{m_\nu u(z_\nu)} \left[A_2(z_\nu) A_3(z_\nu) + A_4(z_\nu) \right] + A_5(z_\nu) \quad ,
\end{aligned}
\tag{5.35}
$$

etc. which gives us another class of IFs dependent explicitly on m_ν, the multiplicity of the zero α_ν. Note that these equations have been verified using a Matlab program, see §C.6.1 on page 190.

5.4.1 A Class of One-point IFs for Multiple Zeros

The first four members of this class are given below. Again, they are displayed in *polynomial* format.

Louis Rall's second-order modified Newton IF [Ral66]

$$\hat{z}_\nu = z_\nu - m_\nu u(z_\nu) \tag{5.36}$$

Joseph Traub's third-order IF [Tra64, p. 139]

$$\hat{z}_\nu = z_\nu + m_\nu \left(\frac{m_\nu - 3}{2} \right) u(z_\nu) - m_\nu^2 A_2(z_\nu) u^2(z_\nu) \quad , \tag{5.37}$$

which might be better known in its *rational* format as Ljiljana Petković, Miodrag Petković, and Dragan Živković have demonstrated that this family is actually a form of Laguerre's method [PPŽ03, pp. 111–112]. For further information about Laguerre's method, especially in the case of real zeros, we recommend Alston Householder's description in [Hou70, pp.176–179].

Joseph Traub's fourth-order IF [Tra64, p. 139]

$$\begin{aligned} \hat{z}_\nu = z_\nu &- m_\nu \left(\frac{m_\nu^2 - 6m_\nu + 11}{6} \right) u(z_\nu) \\ &+ m_\nu^2 (m_\nu - 2) A_2(z_\nu) u^2(z_\nu) \\ &- m_\nu^3 [2A_2^2(z_\nu) - A_3(z_\nu)] u^3(z_\nu) \end{aligned} \quad , \tag{5.38}$$

which he originally defined in a variation of the polynomial form and we refer to as the **Horner** form after the well-known Horner's rule for efficient evaluation of a polynomial [Hou70, pp. 3–4].

Joseph Traub's fifth-order IF [Tra64, p. 139]

$$\begin{aligned} \hat{z}_\nu = z_\nu &+ m_\nu \left(\frac{m_\nu^3 - 10m_\nu^2 + 35m_\nu - 50}{24} \right) u(z_\nu) \\ &- m_\nu^2 \left(\frac{7m_\nu^2 - 30m_\nu + 35}{12} \right) A_2(z_\nu) u^2(z_\nu) \\ &+ m_\nu^3 \left(\frac{3m_\nu - 5}{2} \right) [2A_2^2(z_\nu) - A_3(z_\nu)] u^3(z_\nu) \\ &- m_\nu^4 [5A_2^3(z_\nu) - 5A_2(z_\nu) A_3(z_\nu) + A_4(z_\nu)] u^4(z_\nu) \end{aligned} \quad , \tag{5.39}$$

which he originally defined in Horner form.

5.4.2 A Class of Simultaneous IFs for Multiple Zeros

From Equation (5.30) on page 56 and Equation (2.11) on page 7 we have

$$U(z_\nu) = \epsilon_\nu [1 - B_2(z_\nu)\epsilon_\nu + B_3(z_\nu)\epsilon_\nu^2 - \dots] \quad , \tag{5.40}$$

and since

$$\begin{aligned} U(z_\nu) &= \frac{m_\nu}{\displaystyle\sum_i \frac{m_i}{z_\nu - \alpha_i}} \quad , \\ &= \frac{\epsilon_\nu}{1 + \dfrac{\epsilon_\nu}{m_\nu} S_1(z_\nu)} \quad , \end{aligned} \tag{5.41}$$

we obtain

$$1 + \frac{\epsilon_\nu}{m_\nu} S_1(z_\nu) = [1 - B_2(z_\nu)\epsilon_\nu + B_3(z_\nu)\epsilon_\nu^2 - \dots\ ]^{-1} \quad , \tag{5.42}$$

yielding the first-order approximation

$$\frac{1}{m_\nu} S_1(z_\nu) = B_2(z_\nu) + O(\epsilon_\nu) \quad . \tag{5.43}$$

Noting that

$$S_k'(z_\nu) = -k S_{k+1}(z_\nu) \quad , \tag{5.44}$$

we can successively differentiate Equation (5.42) on page 59, using the following

$$B_i'(z_\nu) = (i+1) B_{i+1}(z_\nu) - 2B_2(z_\nu) B_i(z_\nu) \quad , \tag{5.45}$$

to obtain the following sequence of first-order approximations to $S_k(z_\nu)$.

$$\begin{aligned} \frac{1}{m_\nu} S_1(z_\nu) &= B_2(z_\nu) + O(\epsilon_\nu) \quad , \\ \frac{1}{m_\nu} S_2(z_\nu) &= B_2^2(z_\nu) - 2B_3(z_\nu) + O(\epsilon_\nu) \quad , \\ \frac{1}{m_\nu} S_3(z_\nu) &= B_2^3(z_\nu) - 3B_2(z_\nu)B_3(z_\nu) + 3B_4(z_\nu) + O(\epsilon_\nu) \quad , \\ \frac{1}{m_\nu} S_4(z_\nu) &= B_2^4(z_\nu) - 4B_2^3(z_\nu)B_3(z_\nu) + 2B_3^2(z_\nu) + 4B_2(z_\nu)B_4(z_\nu) \\ &\qquad - 4B_5(z_\nu) + O(\epsilon_\nu) \quad , \end{aligned} \tag{5.46}$$

etc. From Equation (2.11) and Equation (2.13) on page 7 we have

$$S_k(z_\nu) = T_k(z_\nu) + O(\epsilon_\nu) \quad , \tag{5.47}$$

to yield the following sequence of first-order approximations to $T_k(z_\nu)$ which are used subsequently,

$$\begin{aligned}
\frac{1}{m_\nu}T_1(z_\nu) &= B_2(z_\nu) + O(\epsilon_\nu) \quad , \\
\frac{1}{m_\nu}T_2(z_\nu) &= B_2^2(z_\nu) - 2B_3(z_\nu) + O(\epsilon_\nu) \quad , \\
\frac{1}{m_\nu}T_3(z_\nu) &= B_2^3(z_\nu) - 3B_2(z_\nu)B_3(z_\nu) + 3B_4(z_\nu) + O(\epsilon_\nu) \quad , \\
\frac{1}{m_\nu}T_4(z_\nu) &= B_2^4(z_\nu) - 4B_2^3(z_\nu)B_3(z_\nu) + 2B_3^2(z_\nu) + 4B_2(z_\nu)B_4(z_\nu) \\
&\quad - 4B_5(z_\nu) + O(\epsilon_\nu) \quad .
\end{aligned} \tag{5.48}$$

Replacing $B_{\rho-1}(z)$ in R_ρ by the order-preserving approximation involving $T_{\rho-2}(z)$ we generate a class of IFs using simultaneous approximations to all the zeros.

Using Equation (5.35) we obtain a class of IFs, dependent explicitly on m_ν, $\nu = 1,\ 2,\ \ldots,\ N$. The first three members are given below.

Second-order IF

Once again, we know of no polynomial form for the second-order IF. The rational form, derived by Louis Rall, is Equation (B.9) on page 122.

Mick Farmer and George Loizou's third-order IF

$$\hat{z}_\nu = z_\nu - m_\nu u(z_\nu) - m_\nu T_1(z_\nu)u^2(z_\nu) \quad . \tag{5.49}$$

The rational form, derived by Louis Ehrlich, is Equation (B.15) on page 123.

Mick Farmer and George Loizou's fourth-order IF

$$\begin{aligned}
\hat{z}_\nu = z_\nu &- \frac{m_\nu}{8}[3m_\nu^2 - 10m_\nu + 15]u(z_\nu) \\
&+ \frac{m_\nu^2}{2}[3m_\nu - 5]A_2(z_\nu)u^2(z_\nu) \\
&- \frac{m_\nu^2}{2}[3m_\nu A_2^2(z_\nu) + T_2(z_\nu)]u^3(z_\nu)
\end{aligned} \quad . \tag{5.50}$$

The rational form is Equation (B.16) on page 124.

Mick Farmer and George Loizou's fifth-order IF

$$
\begin{aligned}
\hat{z}_\nu = z_\nu &- \frac{m_\nu}{12}[m_\nu^3 + 3m_\nu^2 - 17m_\nu + 25]u(z_\nu) \\
&- \frac{m_\nu^2}{6}[m_\nu^2 - 12m_\nu + 17]A_2(z_\nu)u^2(z_\nu) \\
&+ m_\nu^3\{m_\nu[3A_2^2(z_\nu) - 2A_3(z_\nu)] - 5A_2^2(z_\nu) + 3A_3(z_\nu)\}u^3(z_\nu) \\
&+ \frac{3m_\nu^3}{4}\{4m_\nu[-4A_2^3(z_\nu) + 3A_2(z_\nu)A_3(z_\nu)] - T_3(z_\nu)\}u^4(z_\nu)
\end{aligned} \quad . \tag{5.51}
$$

The rational form is Equation (B.17) on page 124.

Following the same reasoning applied to our simultaneous IFs for simple zeros at the end of section §5.3.2, starting on page 51, the above class of IFs are computed in *parallel*. Convergence can be improved by computing the zeros in a *serial* way, i.e. using the new approximations immediately they become available. As an example, the third-order IF given in Equation (5.49) on page 60 is re-written as

$$
\hat{z}_\nu = z_\nu - m_\nu u(z_\nu) - m_\nu \left[\sum_{i<\nu} \frac{m_i}{z_\nu - \hat{z}_i} + \sum_{i>\nu} \frac{m_i}{z_\nu - z_i} \right] u^2(z_\nu) \quad . \tag{5.52}
$$

The formal term for the study of this technique is called **R-order convergence**. This has already been mentioned in the work of Miodrag Petković, starting with Equation (4.23) on page 32, and will be alluded to again in a section of our chapter on further work, namely §9.4 starting on page 105.

5.4.3 A Class of Variable-order IFs for Multiple Zeros

This is the multiple version of our variable-order IF presented in Equation (5.28) on page 55.

$$
\hat{z}_\nu = z_\nu - m_\nu u(z_\nu) - m_\nu \left[\sum_{i \neq \nu} \frac{m_i}{z_\nu - [\hat{z}_i = z_i - m_i u(z_i)]} \right] u^2(z_\nu) \quad . \tag{5.53}
$$

See §5.3.3 starting on page 54 and §5.6.2 starting on page 64 for arguments that this is a Multipoint IF.

5.5 The Algorithms

Figure 5.5 shows the control and data flows between the different components of our algorithms. A solid arrow indictates control flow and a dashed arrow

indicates data flow. Further details concerning the individual algorithms follow.

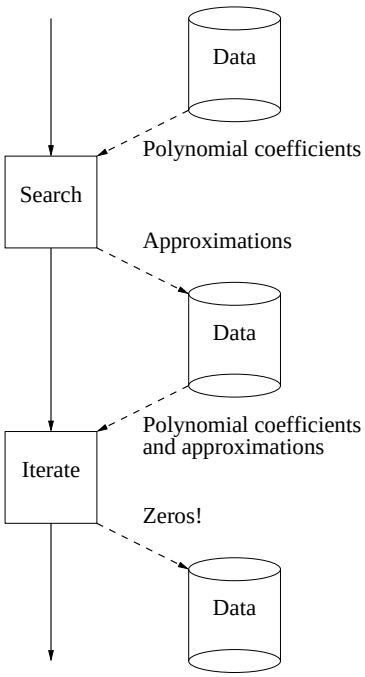

Figure 5.5: Flow Diagram

The algorithms generate standard output so their progress can be monitored on the computer terminal.

5.5.1 Stage 1 – Search

This search stage locates initial approximations to the zeros of $p(z)$ together with their multiplicities. To help the reader relate this to the C code, actual variable and constant names are **emboldened**.

Algorithm 5.3. *This algoritm consists of five steps.*

1. *Use Dirk Dekker's Equation (3.1) in §3.2 on page 14 to locate a circle containing all the zeros of* $\mathbf{myp}(z)$.
2. *Locate the smallest square containing the above circle as illustrated in Figure 3.1 in §3.2 on page 14. This is our initial inclusion region.*
3. *For* $\mathbf{iter1} = 1,\ 2,\ \ldots,\ \mathbf{max1}$
   (a) *For* $\mathbf{iter2} = 1,\ 2,\ \ldots,\ \mathbf{max2}$

i. *Cover each inclusion region (square) with* **sides** $\times$ **sides** *smaller squares.*

ii. *Use Morris Marden's Theorem 3.6 in §3.10 on page 20 to determine the number of zeros in the circle enclosing each smaller square. Retain only those squares containing zeros, each of radius* **oldrad**.

(b) *For each retained square, cover with a larger square of radius* **newrad**. *Drop adjacent smaller squares or nearby smaller squares covered within this new radius.*

(c) *For each of these larger squares, compare the Morris Marden zero count, Theorem 3.6 in §3.10 on page 20 with Jean-Louis Lagouanelle's Equation (3.5) in §3.8 on page 17 for the limit of the multiplicity, and note if they* ***agree****.*

4. *When we complete the above loop we check that our two tests agreed on the multiplicities. If we could not agree on the multiplicities of the approximations then the search stage has failed.*

5. *Otherwise, the approximations to the zeros and their multiplicities agree, so move on to the iterative stage.*

5.5.2 Stage 2 – Iterate

This iterative stage uses our IFs to improve the approximations to the zeros of $p(z)$. Each of the IFs described in §5.4, starting on page 55, is a parameter to the following algorithm. Once again, C program variable and constant names are **emboldened**.

Algorithm 5.4. *This algorithm consists of a single loop containing two steps. The first step contains a further two steps.*

For **iter** $= 1\ , 2,\ \ldots,$ **MAX_ITER** *do*

1. *For each approximation* $\mathbf{z}_\nu$

   (a) *Compute a better new approximation* $\hat{z}_\nu$ *from the old approximations* $\mathbf{z_i},\quad i = 1,\ 2,\ \ldots,$ $\mathbf{ZEROS(z)}$.

   (b) *Compute whether the new approximation is within the required tolerance.*

2. *Exit the above loop if all new approximations are within the required tolerance.*

End do

5.6 Alternative Approaches

This section introduces some alternative methods for generating classes of IFs, often with specific properties.

5.6.1 Derivative-free IFs

It is worth noting here that there is a growing interest in obtaining IFs that are derivative-free. A good example is Kiril Dochev's and Byrnev's rational second-order IF in Equation (B.5) on page 121. Other examples are found in the paper by Sanjay Khattri and Torgrim Log [KL11] that uses a finite difference approximation of the first derivative and a tangential approximation of the second derivative to obtain derivative-free versions of Halley's third-order IF, Equation (5.14) on page 50.

A paper by Changbum Chun [Chu07] also obtains some versions of Halley's third-order IF, Equation (5.14) on page 50, that is free from the second derivative. It also contains references to other work in this area.

Obviously this thesis concentrates on our current research concerning only one-point IF. However, this approach is flagged up as something worth investigating in the future, see §9.3, starting on page 105.

5.6.2 Multipoint IFs

According to Joseph Traub [Tra64, pp. 11–13] the *informational efficiency*, EFF of an IF, is the order of the IF, i.e. ρ in our notation, divided by the *informational usage* of the IF, i.e. d in our case, namely the number of new polynomial and derivative evaluations required per iteration. Thus

$$EFF = \frac{\rho}{d} \quad . \tag{5.54}$$

In addition, he also proves that $EFF \leq 1$ for one-point IFs. This leads him to define a one-point IF as *optimal* if its $EFF = 1$.

Multipoint IFs, as the name implies, are IFs that evaluate polynomials and their derivatives at more than one point. This can produce an IF with an $EFF \geq 1$.

Joseph Traub [Tra64, pp.158–208] devotes two whole chapters to a thorough discussion of multipoint IFs.

Our own variable-order IFs, described in §5.3.3 starting on page 54 (simple) and §5.4.3 starting on page 61 (multiple), are complex examples of multipoint IFs.

5.7 Computational Complexity

REMARK 6. *Lai point 3. Winkler point 7.*

It is possible at this point to ask which of the various IFs is the most efficient in computational terms. This is often measured as the number of floating point operations, aka FLOPS, required.

Since all the IFs presented here have been written in C using the GMP package [Gra11], it is easier to count the number of calls to the **real** arithmetic functions provided by GMP, i.e. addition, subtraction, multiplication, division and square root. Table 5.1 provides the results for our third-order IFs together with our new variable-order IF.

IF	GMP Real Operations
ehrlich3	$34n + 22$
farmer3	$34n + 29$
hansen3	$32n + 31$
traub3	$32n + 25$
farmerv	$N(16n + 35) + 16n + 29$

Table 5.1: Computational Complexity

In Table 5.1, N is the number of distinct zeros of the polynomial and n is the degree of the polynomial. The expressions in the right-hand column are the number of operations required *per approximation per iteration*. Thus **hansen3** and **traub3** are slightly more efficient than **ehrlich3** and **farmer3**. In addition **traub3** is slighlty less complex than **hansen3**.

In contrast, our variable-order IF **farmerv** requires more operations per iteration as it computes N applications of **newton2** for each approximation, but obviously converges faster as the number of approximations increases.

Chapter 6

Database of Test Polynomials

This chapter lists the polynomials we have used to test our algorithms. Where the original (or other given) source is known, then this is acknowledged. If the zeros are known, then each polynomial is expressed as a product of factors.

The degrees of the polynomials vary from two (some simple tests) to 400 in order to provide a wide range of test material. Additionally, the multiplicites of the zeros vary from one (simple zeros) to 40 (see polynomial **164a** in §6.3, starting on page 68) for further details.

Unfortunately, we have gleaned many polynomials over the years from other works where we have not recorded the reference. These polynomials are in the unattributed section, §6.28, starting on page 78.

Each polynomial has a unique identifier consisting of a three-digit number (the degree of the polynomial) followed by a single disambiguating letter. In this chapter the polynomials are grouped by original author(s) (where known). Where appropriate, a second identifier in parentheses is an identifier taken from the original paper. For example, see Dario Bini's polynomials [BF00b, pp. 11–15], Zhonggang Zeng's polynomials [Zen04, pp. 232–235], etc., from which some of this list has been compiled. Some form of further reference may follow which specifies the original source where known, e.g. an explicit reference.

Table 6.1 on page 82 can be consulted to locate individual polynomials by degree.

Many researchers have described, what we would call *derived*, versions of James Wilkinson's famous polynomial, see §6.29 on page 80, but they all describe poly-

nomials with *positive* real zeros, whereas James Wilkinson's example actually has *negative* real zeros [Wil59a, p. 152]. However, we do not consider that this makes any difference to any analysis of the above-mentioned polynomial.

6.1 Oliver Aberth

This polynomial can be found in [Abe73, p. 342].

005a

$$(z - 1 \pm 2i)(z - 2)(z - 3 \pm i)$$

6.2 Milton Abramowitz and Irene Stegun

These are the Laguerre polynomials. They can be found in many publications, including [AS70, pp. 799–780]. The coefficients of these polynomials were generated using the recurrence relation given in [AS70, p. 783].

005g Degree 5

$$z^5 - 25z^4 + 200z^3 - 600z^2 + 600z - 120$$

010j Degree 10

$$z^{10} - 100z^9 + 4050z^8 - 86400z^7 + 1058400z^6 - 7620480z^5 + 3175200z^4 - 72576000z^3 + 81648000z^2 - 36288000z + 3628800$$

The following polynomials are long and have large magnitude coefficients. They are not listed explicitly to prevent padding this thesis with rows of random integers. They were generated using the above-mentioned recurrence relation in order to retain the full accuracy of the integer coefficients. This program is listed in §C.4.4 on page 180.

020k Degree 20

030e Degree 30

050c Degree 50

100c Degree 100

150b Degree 150

Note that, in the past, we used to compute the zeros of low-degree Laguerre polynomials (up to degree 20) by providing lower and upper bounds to their zeros as described in [FL73] and [FL85a].

6.3 Dario Bini and Giuseppe Fiorentino

These polynomials can be found in [BF00b, pp. 11–15].

003r (what would be spiral3)
$(z+1)(z+1+a)(z+1+a+a^2), \quad a = \mathbf{i}/1000$

See Table 2.1 on page 5 for the semantics of **i**.

007k (kam1_1)
$(z - 3 \cdot 10^{-12})^2 + 10^{-6} i z^7$

007l (what would be mandel7)
$(z+1.9408)(z+1.3107)(z+1)(z+0.15652 \pm 1.0322i)(z-0.28227 \pm 0.53006)$

009h (kam2_1)
$(c^2 z^2 - 3)^2 + \mathbf{i} c^2 z^9, \quad c = 10^6$

010l (geom1_10)
$(z+1)(z+a)(z+a^2)\ldots(z+a^9), \quad a = 100\mathbf{i}$

010m (spiral10)
$(z+1)(z+1+a)(z+1+a+a^2)\ldots(z+1+a+a^2+\ldots+a^9), \quad a = \mathbf{i}/1000$

014c (kam4)
$z^{14} + 2 \cdot 10^{24} z^{11} + 10^{48} z^8 + 4z^7 - 4 \cdot 10^{24} z^4 + 4$

020o (lar1)
$z^{20} + 10^{300} z^{14} + z^5 + 1$

031a (mandel31)
This polynomial was constructed using the recurrence relation found in [BF00b, p. 5]. For instance, the Mandelbrot polynomial $p_k(z)$ of degree $n = 2^k - 1$ is defined as

$$\begin{aligned} p_0(z) &= 1 \\ p_i(z) &= z[p_{i-1}^2(z)] + 1 \quad , \quad i = 1, 2, \ldots, k \quad , \end{aligned} \tag{6.1}$$

where the program to generate these polynomials is listed in §C.4.5 on page 182.

040b (wilk40)
$(z-1)(z-2)\dots(z-40)$

044b (kir1_10)
$(z^4-\frac{1}{16})^n[z^4-(\frac{1}{2}+\epsilon)^4], \quad n=10, \quad \epsilon=\frac{1}{4096}$

050e (nroots50)
$z^{50}-1$

052a (lsr4_1)
$(z^{50}+1)(z^2+az+\dfrac{1}{a}), \quad a=10^{10}$

100e (easy100)
$\displaystyle\sum_{j=0}^{100}(1+j)z^j$

164a (kir1_40)
$(z^4-\frac{1}{16})^n[z^4-(\frac{1}{2}+\epsilon)^4], \quad n=40, \quad \epsilon=\frac{1}{4096}$

200a (easy200)
$\displaystyle\sum_{j=0}^{200}(1+j)z^j$

400a (easy400)
$\displaystyle\sum_{j=0}^{400}(1+j)z^j$

6.4 Luigi Brugnano and Donato Trigiante

These polynomials can be found in [BT95, pp. 218–219].

014b (bt02)
$(z-1)^{10}(z-2)^2(z\pm i)$

015f (bt01)
$(z-1)^6(z+1)^2(z\pm i)^3(z-2)$

017b (bt04)
$(z-1)^3(z+1)^4(z-0.5\pm i)^3(z-0.5\pm 0.5i)^2$

020i (bt03)
$(z^2+1)^5(z\pm 0.5i)^4(z\pm 0.75i)$

6.5 Donna Dunaway

These polynomials can be found in [Dun74, pp. 1100–1102].

005t

$(z+1)(z-1.50016\pm 3.57064)^2$

012b

$(z^6-2^6)(z+2)(z^2+3^2)(z-1)^3$

012c

$(z-0.39)^4(z-0.4)^4(z+0.2)^4$

015h

$(z-1)^3(z^2-2)^5(z-3)^2$

016d

$(z\pm 1.7)^4(z\pm 1.3)^4$

020e Scaled version of Wilkinson's polynomial, §6.29 on page 80. To quote from Donna's paper, concerning this scaling.

> "This in effect *squeezes* together the extreme coefficients of $p(z)$, but ignores the behaviour of the rest."

$$0.1e1z^{20} - 0.2528791827178865e2z^{19} + 0.2989309677751350e3z^{18}$$
$$- 0.2194647643162539e4z^{17} + 0.1121321100419947e5z^{16}$$
$$- 0.4234265940941486e5z^{15} + 0.1224852215667365e6z^{14}$$
$$- 0.2776145050337197e6z^{13} + 0.5000616127424668e6z^{12}$$
$$- 0.7218655094001482e6z^{11} + 0.8382839890616433e6z^{10}$$
$$- 0.7830113839688772e6z^{9} + 0.5859739597035374e6z^{8}$$
$$- 0.3485340815756295e6z^{7} + 0.1626647723686615e6z^{6}$$
$$- 0.5843954115543346e5z^{5} + 0.1571225021217127e5z^{4}$$
$$- 0.3029732648207197e4z^{3} + 0.3912780748756267e3z^{2}$$
$$- 0.2987694968616923e2z + 0.1000000706626232e1$$

So this scaling reduces the maximum magnitude of the coefficients from 10^{20}, see those coefficients in Equation (7.1) on page 96, to 10^6.

6.6 Josef Dvorčuk

These polynomials can be found in [Dvo69, pp. 78–79].

010a Table 2

$(z-1)^4(z-2)^3(z-3)^2(z-4)$

020b Table 1. This polynomial is also discussed in depth by Joseph Traub [Tra64, pp. 272–275].
$z^{20} - 1$

6.7 Eulerian Polynomials

These were originally proposed by Leonhard Euler. Their properties are discussed in many publications, including [MW08, p. 1]. The program to generate the coefficients of these polynomial is listed in §C.4.1 starting on page 176.

005u
$z^5 + 26z^4 + 66z^3 + 26z^2 + z + 0$

010k
$z^{10} + 1010z^9 + 47840z^8 + 455192z^7 + 1310354z^6 + 1310354z^5 + 455192z^4 + 47840z^3 + 1010z^2 + z + 0$

020l
Degree is 20.

050d
Degree is 50.

100d
Degree is 100.

6.8 Mick Farmer and George Loizou

These polynomials are modifications of some of our original test polynomials [FL75, p. 254].

020g (fl02), $k = 2$
$(z-1)^{4k}(z-2)^{3k}(z-3)^{2k}(z-4)^k$

030b (fl03), $k = 3$
$(z-1)^{4k}(z-2)^{3k}(z-3)^{2k}(z-4)^k$

6.9 Irene Gargantini and Peter Henrici

This polynomial is Equation (4.13) in [GH72, p. 314].

008a

$z^8 + 2.9742z^7 - 6.7676z^6 + 6.2608z^5 - 54.215z^4 - 97.167z^3 + 7.008z^2 + 7.713z - 15.806$

6.10 Gerald Garside et al.

This polynomial can be found in [GJM68, p. 90].

016a

$(z + \frac{1}{2} \pm \frac{\sqrt{7}}{2})^4 (z + \frac{1}{2} \pm \frac{\sqrt{11}}{2})^4$

6.11 Stefan Goedecker

These are the so-called Fibonacci polynomials, $z^n - z^{n-1} - \cdots - 1$, found in [Goe94, p. 1061].

005q (fib05) $n = 5$

$z^5 - z^4 - z^3 - z^2 - z - 1$

010i (fib10) $n = 10$

020j (fib20) $n = 20$

030d (fib30) $n = 30$

050b (fib50) $n = 50$

100b (fib100) $n = 100$

150a (fib150) $n = 150$

6.12 Stef Graillat et al.

This polynomial can be found in [GLL09, p. 193].

016f

$(0.75 - z)^5(1 - z)^{11}$

6.13 Eldon Hansen and Merrell Patrick

This polynomial can be found in [HP77, p. 266].

006e

$(z + 6)(z - 2)(z - 1 \pm i)(z - 3 \pm 4i)$

6.14 M Igarashi and T Ypma

These polynomials can be found in [IY95, p. 104].

003n (igyp00)

$(z - 2.35)(z - 2.37)(z - 2.39)$

004r (igyp01)

$(z - 2.35)^3(z - 2.56)$

This polynomial can be found in [IY95, p. 106].

010h (igyp02a) $m = 8$

$(z - 10 - 10i)^m(z + 1)^{10-m}$

6.15 Anton Iliev

These polynomials can be found in [Ili00].

006o (iliev00)

$(z - 1)(z + 2)^2(z - 3)^3$

012d (iliev01)

$(z - 1)^2(z - 2)^4(z - 3)^6$

024b (iliev02)

$(z - 1)^4(z - 2)^8(z - 3)^{12}$

048a (iliev03)

$(z - 1)^8(z - 2)^{16}(z - 3)^{24}$

6.16 Misako Ishiguro

These polynomials can be found in [Ish72, p. 49].

003g
$(z+1)^3$

004l
$(z^2-z+1)^2$

004m
$(z^2+z+1)^2$

004o
$(z^2+2)^2$

005f
$(z-1)^2(z+1)(z^2+2)$

006g
$(z^2+1)^2(z^2+2)$

006h
$(z-2)(z+2)(z^2+2)^2$

007e
$z^2(z-1)(z^2+z+1)^2$

008d
$(z^2+z+1)^2(z^2-z+1)^2$

008e
$(z^2+2)^2(z^2-z+1)(z^2+1)$

008f
$(z^2+2)^3(z^2-z+1)$

6.17 Michael Jenkins and Joseph Traub

These polynomials can be found in [JT75].

003l (jt01a), $a = 10^{10}$
$(z \pm a)(z-1)$

003m (jt01b), $a = 10^{-10}$
$(z \pm a)(z-1)$

003o (jt10a), $a = 10^3$

$(z-a)^2(z-\frac{1}{a})$

003 (jt10b), $a = 10^6$

$(z-a)^2(z-\frac{1}{a})$

003q (jt10c), $a = 10^9$

$(z-a)^2(z-\frac{1}{a})$

005m (jt03)

$(z-0.1)(z-0.001)(z-0.00001)(z-0.0000001)(z-0.000000001)$

005n (jt06)

$(z-0.1)(z-1.001)(z-0.998)(z-1.00002)(z-0.99999)$

005r (jt08)

$(z+1)^5$

006n (jt04)

$(z-0.1)^3(z-0.5)(z-0.6)(z-0.7)$

007h (jt07a), $a = 10^{-10}$

$(z-0.001)(z-0.01)(z-0.1)(z-0.1\pm ai)(z-1)(z-10)$

007i (jt07b), $a = 10^{-9}$

$(z-0.001)(z-0.01)(z-0.1)(z-0.1\pm ai)(z-1)(z-10)$

007j (jt07d), $a = 10^{-7}$

$(z-0.001)(z-0.01)(z-0.1)(z-0.1\pm ai)(z-1)(z-10)$

010f (jt05)

$(z-0.1)^4(z-0.2)^3(z-0.3)^2(z-0.4)$

017a (jt02)

$(z-1)(z-2)\dots(z-17)$

6.18 Bin Li et al.

These polynomials can be found in [LNZ08, p. 203].

004s

$(z-0.3)(z+4.6)(z-1.45)(z+10)$

006q

$(z-0.301)(z+4.592)(z-1.458)(z-0.6)(z-15)(z+2)$

6.19 Shi-Mei Ma and Yi Wang

These are the Eulerian polynomials introduced in [MW08, pp. 1–2]. They have only real zeros.

6.20 Fadi Malek and Rémi Vaillancourt

This polynomial can be found in [MV95, p. 9]. It is also presented in [JT75, p. 30].

020n

$(z^{10} - 10^{-20})(z^{10} - 10^{20})$

6.21 Taketomo Mitsui

This polynomial can be found in [Mit83, p. 252].

005h

$(z - 1)(z + 1 \pm \sqrt{7}i)^2$

6.22 Tsuyako Miyakoda

This polynomial can be found in [Miy93, p. 363].

016e

$(z^2 + z + 1)^4(z^2 + z + 3)^4$

6.23 Miodrag Petković et al.

The following polynomials can be found in [PM06, p. 314].

009d Example 2.

$(z + 3)(z \pm 1)(z \pm 2i)(z \pm 2 \pm i)$

The following polynomial is also found in the earlier [SP82, p. 11].

010e Example 1

$(z \pm 1 \pm 2i)(z \pm 2)(z \pm i)(z - 3 \pm 2i)$

The following polynomials can be found in [Pet89, pp. 93–123].

008h (henrici), Example 6, p. 123

$(z+4.1)(z+3.8)(z+2.05)(z+1.85)(z-1.95)(z-2.15)(z-3.9)(z-4.05)$

009f (petk02), Example 3, p. 215

$(z \pm 5i)^2(z-1)^2(z+i)^3$

013c (petk01), Example 4, p. 109

$(z+1)^2(z-3)^3(z^2-2z+5)^2(z+i)^4$

The following polynomial can be found in [PM12, p. 78]

013d

$(z+1)^2(z-1 \pm i)^2(z \pm i)^2(z-2)^3$

The following polynomial can be found in [Pet82, p. 629].

015d

$(z-3)((z+1)^3(z^2+4z+5)^2(z^2-4z+5)^2$

The following polynomial can be found in [PRM12, p. 508].

018c

$(z \pm 1 \pm 2i)(z \pm 2)(z \pm i)(z \pm 3 \pm 2i)(z \pm 2 \pm 3i)(z \pm 3i)$

The following polynomial can be found in [PM12, p. 79]

018d

$(z+1)^2(z+2)^3(z-1 \pm i)^2(z \pm i)^2(z-2)^3(z+2-i)^2$

The following polynomials can be found in [PRM12, p. 509].

019b

$(z \pm 1 \pm 2i)(z \pm 2)(z \pm i)(z \pm 3 \pm 2i)(z \pm 2 \pm 3i)(z \pm 3i)(z-3)$

020m

$(z-4)(z+1)(z \pm 2)(z \pm 2i)(z \pm 3i)(z+1 \pm 2i)(z \pm 1 \pm i)$
$(z-2 \pm i)(z-1 \pm 3i)(z \pm 4i)$

6.24 Tomaso Pomentale

This polynomial can be found in [Pom71, p. 201].

005b

$(z+2.09868+0.455i)(z-2)^2(z-1)(z-0.09868-0.455i)$

6.25 Marica Prešić

This polynomial can be found in [Pre73, p. 303].

007d
$(z+3)(z+1)(z-2)(z-\frac{7}{3})(z-3)(z-7)(z-7.5)$

6.26 Li Shengguo et al.

This polynomial can be found in [SXL09, p. 1291].

009g
$(z^3+4z^2-10)^3$

6.27 Frank Uhlig

These polynomials can be found in [Uhl99].

005o (uhlig01), $a = 0.01$
$(z-a^4)(z-a)^4$

005 (uhlig02), $a = 0.001$
$(z-a^4)(z-a)^4$

008i (uhlig05)
$(z+1)^6(z+2)^2$

6.28 Unattributed

These polynomials from our database either have no known origin or (more likely) that origin has been lost in the mists of time. Note that there is little point in listing a polynomial of high degree, i.e. just the coefficients, where the zeros are unknown. Therefore, a great many entries have not been included. This is indicated by ellipses (...).

002a
$(z-1)(z+1)$

002b
$(z-2)(z-3)$

002c

$(z-2i)(z-3i)$

002d

$iz^2-5iz+6i$

003a

$(z-1)(z+\frac{1}{2}\pm\frac{\sqrt{3}}{2}i)$

003b

z^3+3z+1

003c

$(z-1.0001)(z+1)(z+1.0005)$

003d

$(z-1.00006)(z-1)^2$

003e

z^3-3z+1

003f

$(z+2)(z+3)^2$

003h

$(z-1)(z-2)^2$

003i

$(z+1)(z\pm a)$, $a=1.0001$

003j

z^3-z^2+2z+5

003k

$(z-1)(z+1)^2$

004a

$(z\pm1)(z\pm i)$

004b

$(z-1\pm i)(z-3\pm4i)$

004c

$(z-1)(z+2)(z+3)^2$

004d

$(z-3)(z-2)^2(z+1)$

004e

$(z-2)^3(z+1)$

004f

$(z-2)^4$

REMARK 7. *Polynomials removed. Winkler point 9.*

...

008g

The eight zeros of unity.

$(z \pm 1)(z \pm i)(z \pm 0.707107 \pm 0.707107i)$

REMARK 8. *Polynomials removed. Winkler point 9.*

...

030a

Almost a Fibonacci polynomial, except for the $-2z$ term, but we have no further information.

$z^{30} - z^{29} - z^{28} - \cdots - z^2 - 2z - 1$

REMARK 9. *Polynomials removed. Winkler point 9.*

...

6.29 James Wilkinson

016g (Equation (59))

$2.03253121z^{16}+3.4356048z^{15}+25.1783048z^{14}+37.651096z^{13}+128.218748z^{12}$
$+166.44768z^{11}+345.07256z^{10}+378.908z^9+524.327z^8+468.88z^7+443.576z^6$
$+ 304.08z^5 + 190.68z^4 + 89.6z^3 + 32.8z^2 + 8z + 1$

This polynomial is described in [Wil59b, p. 169], but was originally described by Frank Olver in [Olv52].

020d (Equation (10))

$(z+1)(z+2)\dots(z+20)$

This famous polynomial was first described in [Wil59a, p. 152].

6.30 Joab Winkler et al.

This polynomial can be found in [Win11, p. 9].

008j

$(z-1)^5(z-0.6)^3$

These polynomials can be found in [WLH12, pp. 3491–3495].

028a (Example 11.2)

$(z+0.5926)^{11}(z-2.676)^8(z-0.629)^5(z+9.7181)^4$

028b (Example 11.4)

$(z-1.7054)^6(z+3.1923)^5(z+6.7478)^5(z-9.1949)^4$
$(z+0.032719)^4(z-3.102)^2(z+7.62)^2$

029a (Example 11.1)

$(z+7.5947)^6(z-0.63371)^5(z-1.4923)^5(z-5.4862)^4$
$(z+3.3076)^3(z+3.067)^2(z-0.42244)^2(z-2.509)^2$

035a (Example 11.3)

$(z+9.6084)^{13}(z-3.6683)^7(z+2.1059)^6$
$(z-4.0809)^5(z+1.1539)^4$

038a

$(z+0.67547)^4(z-5.7335)^6(z-2.1747)^7$
$(z+9.5568)^{10}(z+6.5553)^{11}$

6.31 Zhonggang Zeng

This polynomial can be found in [Zen04, p. 219].

010g

$(z-1)^5(z-2)^3(z-3)^2$

These polynomials can be found in [Zen04, pp. 224–227].

012c (twin01), $k = 4$

$(z-0.39)^k(z-0.4)^k(z+0.2)^k$

This polynomial was originally given by Donna Dunaway, see §6.5 on page 70.

015e (triple01), $(m,\ n,\ k) = (5,\ 5,\ 5)$

$(z-0.9)^m(z-1)^n(z-1.1)^k$

015g (fib15)

The Fibonacci polynomial of degree 15, see §6.11 on page 72 for details.

020h (large01)

$(z-1)(z-1.2)(z+1\pm0.3i)(z\pm0.9\pm0.4i)(z+0.7\pm0.7i)$
$(z\pm0.4\pm0.9i)(z\pm1.1i)(z-0.6\pm0.6i)(z\pm0.8i)$

024a (twin02), $k = 8$
$(z - 0.39)^k(z - 0.4)^k(z + 0.2)^k$

030c (triple02), $(m,\ n,\ k) = (10,\ 10,\ 10)$
$(z - 0.9)^m(z - 1)^n(z - 1.1)^k$

036b (twin03), $k = 12$
$(z - 0.39)^k(z - 0.4)^k(z + 0.2)^k$

040a (large02)
The square of **020h**.

050a (fl05)
$(z - 1)^{20}(z - 2)^{15}(z - 3)^{10}(z - 4^5)$

060a (near01), $\epsilon = 0.1$
$(z - 1 - \epsilon)^{20}(z - 1)^{20}(z + 0.5)^{20}$

060b
The sixth power of **010g**
$(z - 1)^{30}(z - 2)^{18}(z - 3)^{12}$

080a (large03)
The square of **040a**.

The following Table 6.1 provides a quick cross-reference to our polynomials and the sections in which they are described.

6.32 Table of Polynomials by Degree

Polynomial	Section
002a	§6.28
002b	§6.28
002c	§6.28
002d	§6.28
003a	§6.28
003b	§6.28
003c	§6.28
003d	§6.28
003e	§6.28
003f	§6.28
003g	§6.16
Continued on next page	

Polynomial	Section
003h	§6.28
003i	§6.28
003j	§6.28
003k	§6.28
003l	§6.17
003m	§6.17
003n	§6.14
003o	§6.17
003	§6.17
003q	§6.17
003r	§6.3
004a	§6.28
004b	§6.28
004c	§6.28
004d	§6.28
004e	§6.28
004f	§6.28
004l	§6.16
004m	§6.16
004o	§6.16
004r	§6.14
004s	§6.18
005a	§6.1
005b	§6.24
005f	§6.16
005g	§6.2
005h	§6.21
005m	§6.17
005n	§6.17
005o	§6.27
005	§6.27
005q	§6.11
005r	§6.17
005t	§6.5
005u	§6.7
006e	§6.13
006g	§6.16
006h	§6.16

Continued on next page

Polynomial	Section
006n	§6.17
006o	§6.15
006q	§6.18
007d	§6.25
007e	§6.16
007h	§6.17
007i	§6.17
007j	§6.17
007k	§6.3
007l	§6.3
008a	§6.9
008d	§6.16
008e	§6.16
008f	§6.16
008g	§6.28
008h	§6.23
008i	§6.27
008j	§6.30
009d	§6.23
009f	§6.23
009g	§6.26
009h	§6.3
010a	§6.6
010e	§6.23
010f	§6.17
010g	§6.31
010h	§6.14
010i	§6.11
010j	§6.2
010k	§6.7
010l	§6.3
010m	§6.3
012b	§6.5
012c	§6.5
012d	§6.15
013c	§6.23
013d	§6.23
014b	§6.4

Continued on next page

Polynomial	Section
014c	§6.3
015d	§6.23
015e	§6.31
015f	§6.4
015g	§6.31
015h	§6.5
016a	§6.10
016d	§6.5
016e	§6.22
016f	§6.12
017a	§6.28
017b	§6.17
018c	§6.23
018d	§6.23
019b	§6.23
020b	§6.6
020d	§6.29
020e	§6.5
020g	§6.8
020h	§6.31
020i	§6.4
020j	§6.11
020k	§6.2
020l	§6.7
020m	§6.23
020n	§6.20
020o	§6.3
024a	§6.31
024b	§6.15
028a	§6.30
028b	§6.30
029a	§6.30
030a	§6.28
030b	§6.8
030c	§6.31
030d	§6.11
030e	§6.2
031a	§6.3

Continued on next page

Polynomial	Section
035a	§6.30
036b	§6.31
038a	§6.30
040a	§6.31
040b	§6.3
044b	§6.3
048a	§6.15
050a	§6.31
050b	§6.11
050c	§6.2
050d	§6.7
050e	§6.3
052a	§6.3
060a	§6.31
060b	§6.31
080a	§6.31
100b	§6.11
100c	§6.2
100d	§6.7
100e	§6.3
150a	§6.11
150b	§6.2
164a	§6.3
200a	§6.3
400a	§6.3

Table 6.1: Cross-reference of Polynomials

REMARK 10. *Unattributed polynomials about which we no information have been removed from the above. Winkler point 9.*

Chapter 7

Test Results

This chapter gives a summary of our computational results when applying the search stage together with the IF stage. All the IFs presented in this thesis, found in §5.4, starting on page 55, are applied to all the polynomials described in Chapter 6, starting on page 66.

7.1 Summary of Results

REMARK 11. *Split summary into explicit search and iterate stages. Winkler point 5.*

7.1.1 Search Summary

Table 7.1 on page 88 summarises those polynomials that did not complete the search stage with the default number (8 in our case) of outer iterations.

Those polynomials taking less than the default are probably the result of earlier tinkering with script parameters or our attempts to improve the speed of the search stage.

Those polynomials taking more than the default are because additional iterations were required to resolve clusters of zeros or close multiple zeros.

7.1.2 Iterate Summary

REMARK 12. *Initial paragraph added. Winkler point 6.*

Iterations	Polynomials
3	010h
4	004i 004r 006i
5	003n 005o 006j 007i 015e 048a 400a
6	009a 012a 016f 030c 036b 060a 080a
7	010c 060b
9	100c
11	005m 013a
12	007j 028b 035a 038a 150b
15	020l
16	003m 003r 005t 007h 008j 016d
32	010l
40	050d
48	009h 010m
64	007k 014c
68	100d

Table 7.1: Search Stage Iterations

This section illustrates how our IFs compare with many, comparable, IFs previously discovered. Our third-order IF takes the same number of iterations to converge as the other third-order IFs. Furthermore, our variable-order IF is shown to be superior, i.e. faster, to those others showcased.

Table 7.2, starting on page 89, is a summary of our IF computational results. Each IF is identified by the person we consider was the first to derive that IF. **Rall2** is given in Equation (5.36) on page 57, **Ehrlich3** is given in Equation (B.15) on page 123, **Farmer3** is given by Equation (5.49) given on page 60, **Hansen3** is given in Equation (B.10) on page 122, and **Farmerv** is given in Equation (5.53) on page 61. The name is followed by the notional order of convergence.[1] The IFs were run in *parallel* mode unless followed by **(s)**, which indicates the *serial* mode of the IF.

The search stage either succeeds (`S`) or fails (`F`) to locate suitable initial approximations to the zeros. For the different IFs the table shows the number of iterations taken to converge or `F` if they failed to converge. As expected, the number of iterations is less for IFs with a higher-order of convergence.

[1] Farmerv indicates our new IF where the order of convergence depends on the degree of the polynomial.

Poly	Search	Rall2	Ehrlich3	Ehrlich3 (s)	Farmer3	Farmer3 (s)	Hansen3	Traub3	Farmerv
002a	S	3	3	3	3	3	3	3	3
002b	S	4	3	3	3	3	3	3	3
002c	S	4	3	3	3	3	3	3	3
002d	S	4	3	3	3	3	3	3	3
003a	S	4	4	4	3	3	3	3	3
003b	S	4	4	4	3	3	3	3	3
003c	S	4	3	3	3	3	3	3	3
003d	S	6	4	4	4	4	4	4	5
003e	S	4	3	3	3	3	3	3	3
003f	S	4	3	3	3	3	3	3	4
003g	S	2	2	2	2	2	2	2	2
003h	S	4	3	3	3	3	3	3	4
003i	S	4	4	4	3	3	3	3	3
003j	S	4	4	4	3	3	3	3	3
003k	S	3	3	3	3	3	3	3	3
003l	S	3	3	3	3	3	3	3	2
003m	S	5	4	3	4	4	4	4	4
003n	S	6	4	4	4	4	4	4	4
003o	S	4	3	3	3	3	3	3	4
003p	S	4	3	3	3	3	3	3	4
003q	S	4	3	3	3	3	3	3	4
003r	S	3	3	3	3	3	3	3	2
004a	S	3	3	3	3	3	3	3	2
004b	S	4	4	4	3	3	3	3	2
004c	S	4	3	3	3	3	3	3	3
004d	S	4	3	3	3	3	3	3	3
004e	S	3	3	3	3	3	3	3	3
004f	S	2	2	2	2	2	2	2	2
004g	S	4	4	4	3	3	3	3	3
004h	S	4	3	3	3	3	3	3	3
004i	S	6	5	5	4	4	4	4	4
004j	S	4	3	3	3	3	3	3	3
004k	S	4	4	4	3	3	3	3	2
004l	S	4	4	4	3	3	3	3	4

Continued on next page

Poly	Search	Rall2	Ehrlich3	Ehrlich3 (s)	Farmer3	Farmer3 (s)	Hansen3	Traub3	Farmerv
004m	S	4	4	4	3	3	3	3	4
004n	S	4	3	3	3	3	3	3	4
004o	S	4	3	3	3	3	3	3	4
004p	S	4	3	3	3	3	3	3	2
004q	S	2	2	2	2	2	2	2	2
004r	S	5	4	4	4	4	4	4	5
004s	S	4	3	3	3	3	3	3	3
005a	S	4	4	4	3	3	3	3	2
005b	S	4	4	4	3	3	3	3	2
005c	S	4	4	4	3	3	3	3	2
005d	S	4	4	4	3	3	3	3	2
005e	S	4	3	3	3	3	3	3	3
005f	S	4	3	3	3	3	3	3	2
005g	S	4	3	3	3	3	3	3	2
005h	S	3	3	3	3	3	3	3	2
005i	S	4	4	4	3	3	3	3	2
005j	S	4	3	3	3	3	3	3	3
005k	S	2	2	2	2	2	2	2	2
005l	S	4	4	4	3	3	3	3	2
005m	S	6	4	4	4	4	4	4	2
005n	S	6	4	4	4	4	4	4	2
005o	S	6	4	3	4	4	4	4	4
005p	S	4	3	3	3	3	3	3	3
005q	S	4	4	4	3	3	3	3	2
005r	S	2	2	2	2	2	2	2	2
005s	S	4	4	4	3	3	3	3	2
005t	S	6	6	6	4	4	4	5	2
005u	S	4	3	3	3	3	3	3	2
006a	S	4	4	4	3	3	3	3	2
006b	S	4	4	4	3	3	3	3	2
006c	S	4	4	4	3	3	3	3	2
006d	S	4	4	4	3	3	3	3	2
006e	S	4	4	4	3	3	3	3	2
006f	S	4	4	4	3	3	3	3	2

Continued on next page

Poly	Search	Rall2	Ehrlich3	Ehrlich3 (s)	Farmer3	Farmer3 (s)	Hansen3	Traub3	Farmerv
006g	S	4	3	3	3	3	3	3	2
006h	S	4	3	3	3	3	3	3	2
006i	S	6	6	6	4	4	4	4	2
006j	S	6	7	7	5	5	4	5	2
006k	S	4	4	4	3	3	3	3	2
006l	S	4	3	3	3	3	3	3	2
006m	S	4	3	3	3	3	3	3	2
006n	S	4	3	3	3	3	3	3	2
006o	S	4	3	3	3	3	3	3	2
006p	S	4	3	3	3	3	3	3	2
006q	S	4	3	3	3	3	3	3	2
007a	S	4	4	4	3	3	3	3	2
007b	S	4	5	5	3	3	3	3	2
007c	S	4	4	4	3	3	3	3	2
007d	S	4	3	3	3	3	3	3	2
007e	S	4	4	4	3	3	3	3	2
007f	S	4	4	4	3	3	3	3	2
007g	S	4	3	3	3	3	3	3	2
007h	S	7	7	7	5	5	5	5	2
007i	S	4	4	4	3	3	3	3	2
007j	S	7	7	7	5	5	5	5	2
007k	S	1	1	1	1	1	1	1	1
007l	S	4	4	4	3	3	3	3	2
008a	S	4	4	4	3	3	3	3	2
008b	S	4	4	4	3	3	3	3	2
008c	S	5	5	5	4	4	4	4	2
008d	S	4	4	4	3	3	3	3	2
008e	S	4	4	4	3	3	3	3	2
008f	S	4	4	4	3	3	3	3	2
008g	S	4	4	4	3	3	3	3	2
008h	S	4	3	3	3	3	3	3	2
008i	S	4	3	3	3	3	3	3	3
008j	S	5	5	5	3	3	3	3	2
009a	S	6	7	7	5	5	4	5	2

Continued on next page

Poly	Search	Rall2	Ehrlich3	Ehrlich3 (s)	Farmer3	Farmer3 (s)	Hansen3	Traub3	Farmerv
009b	S	4	4	4	3	3	3	3	2
009c	S	4	4	4	3	3	3	3	2
009d	S	4	4	4	3	3	3	3	2
009e	S	4	3	3	3	3	3	3	2
009f	S	4	4	4	3	3	3	3	2
009g	S	4	4	4	3	3	3	3	2
009h	S	1	1	1	1	1	1	1	1
010a	S	4	3	3	3	3	3	3	2
010b	S	4	4	4	3	3	3	3	2
010c	S	7	6	6	5	5	5	5	2
010d	S	4	4	4	3	3	3	3	2
010e	S	4	4	4	3	3	3	3	2
010f	S	4	3	3	3	3	3	3	2
010g	S	4	3	3	3	3	3	3	2
010h	S	7	6	6	5	5	5	5	7
010i	S	4	4	4	3	3	3	3	2
010j	S	4	3	3	3	3	3	3	2
010k	S	5	4	4	4	4	4	4	2
010l	S	3	3	3	3	3	3	3	2
010m	S	1	1	1	1	1	1	1	1
012a	S	6	6	6	4	4	4	4	2
012b	S	4	4	4	3	3	3	3	2
012c	S	4	3	3	3	3	3	3	2
012d	S	4	3	3	3	3	3	3	2
013a	S	6	6	6	4	4	4	4	2
013b	S	4	4	4	3	3	3	3	2
013c	S	4	4	4	3	3	3	3	2
013d	S	4	4	4	3	3	3	3	2
014a	S	4	4	4	3	3	3	3	2
014b	S	4	4	4	3	3	3	3	2
014c	S	1	1	1	1	1	1	1	1
015a	S	4	4	4	3	3	3	3	2
015b	S	4	4	4	3	3	3	3	2
015c	S	4	3	3	3	3	3	3	2

Continued on next page

Poly	Search	Rall2	Ehrlich3	Ehrlich3 (s)	Farmer3	Farmer3 (s)	Hansen3	Traub3	Farmerv
015d	S	4	4	4	3	3	3	3	2
015e	S	5	4	4	3	3	4	4	3
015f	S	4	4	4	3	3	3	3	2
015g	S	4	4	4	3	3	3	3	2
015h	S	4	3	3	3	3	3	3	2
016a	S	4	4	4	3	3	3	3	2
016b	S	4	4	4	3	3	3	3	2
016c	S	4	4	4	3	3	3	3	2
016d	S	4	4	4	3	3	3	3	2
016e	S	4	4	4	3	3	3	3	2
016f	S	5	3	3	3	3	4	4	3
016g	S	4	4	4	3	3	3	3	2
017a	S	4	3	3	3	3	3	3	2
017b	S	4	4	4	3	3	3	3	2
018a	S	4	4	4	3	3	3	3	2
018b	S	4	4	4	3	3	3	3	2
018c	S	4	4	4	3	3	3	3	2
018d	S	4	4	4	3	3	3	3	2
019a	S	4	4	4	3	3	3	3	2
019b	S	4	4	4	3	3	3	3	2
020a	S	4	4	4	3	3	3	3	2
020b	S	4	4	4	3	3	3	3	2
020c	S	4	4	4	3	3	3	3	2
020d	S	4	3	3	3	3	3	3	2
020e	S	4	3	3	3	3	3	3	2
020f	S	4	4	4	3	3	3	3	2
020g	S	4	3	3	3	3	3	3	2
020h	S	4	4	4	3	3	3	3	2
020i	S	4	3	3	3	3	3	3	2
020j	S	4	4	4	3	3	3	3	2
020k	S	5	3	3	4	4	4	4	2
020l	S	6	6	6	4	4	4	4	2
020m	S	4	4	4	3	3	3	3	2
020n	S	6	6	6	4	4	4	4	2

Continued on next page

Poly	Search	Rall2	Ehrlich3	Ehrlich3 (s)	Farmer3	Farmer3 (s)	Hansen3	Traub3	Farmerv
020o	S	F	F	F	F	F	F	F	F
024a	S	3	2	2	2	2	2	2	2
024b	S	4	3	3	3	3	3	3	2
028a	S	4	3	3	3	3	3	3	2
028b	S	3	3	3	3	3	3	3	9
029a	S	4	3	3	3	3	3	3	2
030a	S	4	4	4	3	3	3	3	2
030b	S	4	3	3	3	3	3	3	2
030c	S	3	2	2	3	3	3	3	2
030d	S	4	4	4	3	3	3	3	2
030e	S	5	5	5	4	4	4	4	2
031a	S	4	4	4	3	3	3	3	2
035a	S	3	3	3	3	3	3	3	2
036a	S	4	4	4	3	3	3	3	2
036b	S	3	2	2	2	2	2	2	2
038a	S	6	6	6	4	4	4	4	2
040a	S	6	6	6	4	4	4	F	2
040b	S	5	5	5	4	4	4	4	2
044a	S	5	5	5	3	3	3	3	2
044b	S	5	4	4	4	4	4	4	2
048a	S	5	3	3	3	3	3	4	3
050a	S	4	3	3	3	3	3	3	3
050b	S	4	4	4	3	3	3	3	2
050c	S	6	6	6	4	4	4	4	2
050d	S	4	4	4	3	3	3	3	2
050e	S	4	4	4	3	3	3	3	2
052a	S	5	5	5	4	4	4	4	2
055a	S	5	5	5	3	3	3	3	2
055b	S	5	5	5	3	3	3	3	2
056a	S	5	5	5	4	4	4	4	2
057a	S	5	5	5	4	4	4	4	2
060a	S	2	2	2	2	2	2	2	2
060b	S	3	2	2	2	2	2	2	2
080a	S	4	4	4	3	3	3	3	2

Continued on next page

Poly	Search	Rall2	Ehrlich3	Ehrlich3 (s)	Farmer3	Farmer3 (s)	Hansen3	Traub3	Farmerv
100a	S	5	5	5	4	4	4	4	2
100b	S	4	4	4	3	3	3	3	2
100c	S	5	5	5	4	4	4	4	2
100d	S	6	5	5	4	4	4	4	3
100e	S	4	4	4	3	3	3	3	2
101a	S	5	5	5	4	4	4	4	2
150a	S	4	4	4	3	3	3	3	2
150b	S	4	4	4	3	3	3	3	2
164a	S	1	1	1	1	1	1	1	1
200a	S	4	4	4	3	3	3	3	2
400a	S	6	6	6	4	4	4	4	2

Table 7.2: Number of Iterations for Convergence

The above results illustrate the values that one expects. Higher-order IFs converge in fewer iterations than lower-order IFs. Once the degree of the polynomial increases, our variable order IF, Equation (5.53) on page 61, is always faster, in terms of the number of iterations.

7.2 Successful Results

Our database of polynomials given in Chapter 6, starting on page 66, contains over two hundred distinct polynomials[2]. Section 7.1, starting on page 87, shows the results when these polynomials were tested on our system.

All the polynomials passed through the search stage successfully, providing good initial approximations for the IFs used in the iterative stage.

Only two polynomials failed to produce the correct result during the iterative stage, and this is discussed in §7.3, starting on page 97. This means that only 0.93% failed.

We consider that this demonstrates the success of our approach to locating the zeros of arbitrary real or complex polynomials.

[2]215 at the last count.

As we pointed out in Chapter 1, starting on page 1, we derived a search stage to prefix our IF algorithms in order to generate computationally our initial approximations, rather than choosing convenient initial approximations.

This search stage raised a number of mini-problems which have not been addressed so far. We now redress this anomaly.

7.2.1 Coefficients of Test Polynomials

We are interested in the approach taken by both Joab Winkler and Zhonggang Zeng (and possibly others) in terms of the *inexact* coefficients of polynomials. The implication is that their IFs will still converge to the "correct" zeros. However, this raises the issue of what is the "correct" polynomial. How far removed do the coefficients have to be before we are considering another, distinct, polynomial?

Our approach is to take each polynomial on its merits, i.e. the coefficients *are* the polynomial. Just that.

Therefore, when we investigate polynomials where the exact values of the coefficients are important (for example, see the Laguerre polynomials in §6.2 on page 67, the Eulerian polynomials in §6.7 on page 71, or James Wilkinson's polynomial in §6.29 on page 80) we compute those coefficients to whatever accuracy is required. Just to emphasise this, Equation (7.1) illustrates the *exact* coefficients of James Wilkinson's polynomial.[3]

$$\begin{aligned} & z^{20} + 210z^{19} + 20,615z^{18} + 1,256,850z^{17} + 53,327,946z^{16} \\ & \quad + 1,672,280,820z^{15} + 40,171,771,630z^{14} + 756,111,184,500z^{13} \\ & \quad + 11,310,276,995,381z^{12} + 135,585,182,899,530z^{11} \\ & \quad + 1,307,535,010,540,395z^{10} + 10,142,299,865,511,450z^{9} \\ & \quad + 63,030,812,099,294,896z^{8} + 311,333,643,161,390,640z^{7} \\ & \quad + 1,206,647,803,780,373,360z^{6} + 3,599,979,517,947,607,200z^{5} \\ & \quad + 8,037,811,822,645,051,776z^{4} + 12,870,931,245,150,988,800z^{3} \\ & \quad + 13,803,759,753,640,704,000z^{2} + 8,752,948,036,761,600,000z \\ & \quad + 2,432,902,008,176,640,000 \end{aligned} \tag{7.1}$$

Note that, just this once, the comma separator has been included in order to emphasis the magnitudes of these coefficients. Also note that James Wilkinson's original polynomial has *negative* real zeros.

[3]This is explicitly included on George's request.

The programs that generate polynomial coefficients from other data are listed in §C.4 starting on page 176.

7.2.2 Multiplicities of Zeros

As described in the text (see §5.5.1 on page 62), the search stage does finish when our two tests agree on the multiplicities of the approximations.

However, this is not always the case. Some polynomials can have clusters of multiple zeros, and this has to be resolved before the search stage can terminate.

Therefore, the program for the search stage can be made to continue until the correct multiplicities have been determined.

7.2.3 Clusters of Zeros

As pointed out in the previous section, some polynomials have clusters of zeros, often appearing as one or more multiple zeros.

Once again, the program for the search stage can be made to continue until the clusters resolve themselves. Quite often, this requires increasing the number of binary digits used in the precision of the representations of the approximations.

This is where the flexibility of the GNU Multiple Precision Arithmetic Library [Gra11] comes into its own. It only takes a moment to change the precision used before re-running the program.

7.3 Failed Results

This section presents and discusses those polynomials that we have been unable to solve using our existing algorithms. In addition, we investigate why they failed, which may help identify what improvements might be introduced. Also see Chapter 9 for more possible improvements.

020o

This polynomial was described by Dario Bini, see §6.3 on page 68 for its definition.

In an exchange of e-mail with Dario he stated that the polynomial had ten complex conjugate pairs as zeros. We replied that our algorithm yielded three complex conjugate pairs and a single real zero of multiplicity 14.

040a

This polynomial was described by Zhonggang Zeng, see §6.31 on page 81 for its definition.

This polynomial is the square of the polynomial **020h**, which has only simple zeros. Three of the tested IFs, **Farmer3**, **Farmer3s**, and **Traub3** fail to converge for one approximate zero only. The other IFs all converge.

The problem occurs with one approximate zero only. Its relative error is consistently of value 4.03×10^{-24} which is slightly larger than our fixed value of ϵ, used to decide on convergence, which is 1×10^{-24}.

Interestingly the square of this polynomial, **080a**, is solved by all the tested IFs.

7.4 Summary of Timings

Table 7.3 on page 98 is a summary of our computational timings. The table headings are the same for those described before Table 7.2 on page 89.

The clock timer on our computer is not fine-grained and can only resolve CPU clock times to hundredths of a second. For details of our hardware, see Table C.1 on page 126. All the timings for polynomials of degree 50 or lower were recorded as `0.00` seconds. The timings shown are the total CPU time to achieve convergence for each of the IFs.

Poly	**Rall2**	**Ehrlich3**	**Ehrlich3 (s)**	**Farmer3**	**Farmer3 (s)**	**Hansen3**	**Traub3**	**Farmerv**
052a	0.04	0.09	0.09	0.07	0.07	0.05	0.05	0.04
100d	0.05	0.14	0.14	0.14	0.13	0.10	0.10	0.06
150b	0.23	0.62	0.62	0.46	0.46	0.35	0.35	0.23
164a	0.02	0.02	0.02	0.02	0.02	0.04	0.03	0.02
200a	0.41	1.10	1.10	0.82	0.82	0.61	0.61	0.42
400a	2.03	5.60	5.60	4.30	4.28	3.17	3.22	2.02

Table 7.3: Timing Comparisons

As expected, the timings increase with the order of convergence of the IFs, because more function evaluations are required. It is interesting that the timings for the third-order IFs show considerable differences. This may be down to C coding inefficiences on our part that have crept in unnoticed.

However, what is most interesting is the timings of our **Farmerv** IF, which is on a par with the second-order IF **Rall2**. This is probably because our variable-order IF given in Equation(5.53) on page 61 uses a great many *Newton corrections*, see [Pet89, p. 85] for details, per iteration.

7.5 Other Zero Finders

REMARK 13. *Winkler points 3 and 10.*

Obviously, we are not alone in developing algorithms for finding the zeros of arbitrary polynomials. Our successful results are compared to three other, well known, zero finders, whose details are given below.

For comparison purposes, we take three polynomials originally from Zhonggang Zeng's database, see §6.31 starting on page 81, one from Joab Winkler's database, see §6.30 starting on page 80, and one from Dario Bini's database, see §6.3 starting on page 68. These are

020h (Zeng)
: A polynomial of degree twenty consisting of simple zeros, mainly as complex conjugate pairs.

035a (Winkler)
: A polynomial with high multiplicity zeros (13, 7, 6, 5, and 4).

040a (Zeng)
: The square of **020h**, providing double zeros.

080a (Zeng)
: The square of **040a**, providing quadruple zeros and a reasonably high degree polynomial.

164a (Bini)
: A polynomial of high degree with four zeros of multiplicity 40 (± 0.5 and $\pm 0.5i$) and a cluster of four simple zeros nearby (± 0.500244141 and $\pm 0.500244141i$).

The results are summarised below.

7.5.1 Matlab's built-in function roots

Documentation about this function can be found in [Mat12]. It found the simple zeros of **020h** only. For the other polynomials, the incorrect zeros were

presented as complex conjugate pairs. We consider that this behaviour is bad, as Matlab users using `roots` as a black box might accept the results as correct.

7.5.2 Dario Bini's MPSolve Package

This package is described in [BF00a]. It found the correct multiple zeros for four of the five polynomials. It failed with **164a**, one of Dario's own examples, when the coefficients were translated from our format (floating point coefficients) to **mpsolve**'s similar format. However, it computed the correct zeros when run with its own chosen format (scaled integer coefficients).

This package's biggest drawback is that it prints the zeros individually, rather than grouping them by multiplicity. This means that it is necessary to inspect the results visually (or via an additional program) in order to derive the multiplicities. In addition, clusters may be hidden if the computed zeros are printed using a truncated format.

7.5.3 Zhonggang Zeng's MULTROOT Package

This package is described in [Zen04]. It found the correct multiple zeros for four of the five polynomials. It failed with **164a** — no computational results printed, just nothing except the polynomial's coefficients.

In addition, there were problems with polynomials having coefficients with non-zero imaginary parts, such as **010l**. We have not had time to investigate further.

Finally, there were rounding errors due to the fact that Matlab only uses double precision arithmetic.

7.5.4 Overall Summary

As demonstrated above, our package computed correctly the zeros of all the polynomials in our database. All of the other, well-known, packages failed in some way on one or more of the polynomials in our database.

Chapter 8

Conclusions

This chapter brings together our thoughts on how successful our approach has been and why we think our work is important. Lastly, we summarise what other researchers are doing in this field and compare their results with ours, which we think will stand the test of time.

8.1 Successful Implementation

Firstly, our seach algorithm has shown that it is robust when handling both polynomials of high degree (up to degree 400) and polynomials with zeros of high multiplicity (up to order 40).

Using circular tests (see §3.10 starting on page 20 for details) when we divide the complex plane into squares has caused minor problems with overlap (see Figure 3.2 on page 19 for details), but our search algorithm has been modified to cope extremely well with this problem.

The fact that our search algorithm never failed to locate appropriate squares containing approximations to the zeros substantiates this fact.

Secondly, our IFs have been run in parallel with those found in the classical literature and demonstrated that they hold their own, especially the simultaneous IFs (see §5.4.2 starting on page 58 for details) and our new variable-order IF (see §5.4.3 starting on page 61 for details).

8.1.1 Fine Tuning

It is not possible to write computer programs that do exectly what we want every time for arbitrary databases without intervention. One obvious example is polynomials that have zeros of high multiplicity need more accuracy (i.e. precision) in their representation of values, especially the zeros themselves. Also, computations need to be carried out with this greater precision.

Another example is the number of iterations taken during the search stage. Some polynomials can isolate the approximations quite quickly, while others need more time (i.e. iterations) to resolve the correct distribution of the approximations.

Other, less obvious, values can also be modified at run time. Where the effect is minimal we have left parameters at their default values (see Table C.3 on page 127 for these options). However, in extremes it is necessary to change these options. As an example, our polynomial of degree 400 took four hours for the search stage to complete using the default number of outer iterations (e.g. 8), but reducing this value to 5 brought the time taken during the search stage to two hours, a much more respectable value.

8.1.2 Computer Power

We think this is an important factor in the way we tackle the problem of locating the zeros of polynomials. The sheer power of our current computers means that we can look at alternatives that were not available to the earlier researchers in this field.

During research for the historic perspective chapter (see Chapter 3 starting on page 13) we looked especially at the example polynomials presented at the time. Their degree was low, i.e. very little above 20, and the multiplicities hardly reached 10.

The reason is obvious to us now — computer power was both expensive and slow. Today, the cost of serious computers is within the reach of individuals, let alone departments with laboratories full of computers, and the raw materials are dead cheap. More importantly, their speed has increased dramatically as can be seen in Table 7.3 on page 98. This is why the machines on our desks easily out-perform those machines that came before.

8.1.3 Parallelisation

It is frustrating that the computers we use have operating systems that do not take account of the CPU cores in each of them, four to be exact in our case. Our current programming languages are all single-threaded.

If possible, the search stage could be distributed amongst different CPUs and the results amalgamated as each set of sub-search results became available.

In addition, the iterative stage could be distributed according to IF.

8.2 Alternative Approaches

One alternative might be a change of hardware. We read in the more general computer literature that Graphical Processing Units (GPUs) process data faster than conventional CPUs. This is because they are used to implement moving computer graphics for computer games (and presumably other applications).

A machine that allowed its GPUs to be programmed with our software should speed up the computations considerably. However, we think that an implementation of Linux, and its supporting software, is not likely to be seen in the very near future.

A second alternative might be a change of software. Two popular computer languages with built-in support for arbitrary precision numbers are Python [Lut08] and Ruby [FM08]. There are other computer languages that support arbitrary precision of some form, but we have not followed these up.

Chapter 9

Further Work

This Chapter presents where we see our research heading in the immediate future.

9.1 Squares Instead of Circles

We have already mentioned the problems encountered with using circles to determine the number of zeros in an enclosed square. See Chapter 8 starting on page 101 and its reference back to Figure 3.2 on page 19 for details. Regions with straight line boundaries should not cause these problem.

Triangles are the simplest, mentioned in [JT09], although we have not followed this up.

Squares, the next best thing, are mentioned in [Neu88] and used in at least one implementation, that by Irene Gargantini [GM67].

9.2 Faster Simultaneous IFs

In §5.4.2, starting on page 58, for an IF of order ρ we replace the highest-order derivative of $p(z_\nu)$, usually in the form of $B_{\rho-1}(z_\nu)$, with a first-order approximation in terms of $T_{\rho-2}(z_\nu)$.

One possible approach would be to replace some of the lower-order derivatives of $p(z_\nu)$ with more complex, higher-order, expressions containing combinations of $A_i(z_\nu)$ and $T_k(z_\nu)$.

The obvious advantage of replacing $B_{\rho-1}(z_\nu)$, with a first-order approximation in terms of $T_{\rho-2}(z_\nu)$ is that we replace the computation of two function calls with a simple summation. Whether this advantage continues with more complicated replacements remains to be seen.

9.3 Derivative-free IFs

There is a long history of derivative-free IFs, mainly because they avoid additional polynomial evaluations, which were considered *expensive* with computers of that time. Joseph Traub [Tra64, pp. 209–214] devotes a chapter to this topic.

We have concentrated on one-point IFs in our research to date, although we should give this topic some consideration in the future.

9.4 R-order Convergence

A formal definition is given in [Mon12]. There are also useful references in [Pet89] and [PPS89]. We will start with these to investigate how we can develop better bounds for the order of convergence of those IFs using this technique.

9.5 Generalised Order of Convergence

In our work to date, we have derived the order of convergence of our IFs by hand, or more recently by using tools such as Matlab [Mat12]. However, our holy grail is to be able to derive closed forms for all of our IFs and, at the same time, the equations defining their orders of convergence.

Joseph Traub [Tra64] has done some of this, as he covered each different form of earlier IFs; however, it would really satisfy us if we could complete the work for our different and newer IFs.

It will be especially important to realise the order of convergence, and therefore the asymptotic error constant, of our simple and multiple simultaneous IFs.

9.6 Newer Tools

When we first started our research in this area there were very few tools available that we could use to make our research easier. Numerical algorithms were

being published in many journals, but the hardware was slow and numerical programming was dominated by the Fortran programming language [McC61].

In the United Kingdom, Algol 60 [B$^+$63] was the language of choice, but there were no supporting numerical libraries (the NAG library [FB77] had only just appeared in October 1971, with Algol 60 and Fortran implementations). As mentioned elsewhere, this meant that, especially for the search stage, we were utilising checkpoint/restart techniques to ensure sufficient computer time was available for the algorithm to terminate.

Since then, the facilities available have multiplied and we are able to accomplish much more in a more user-friendly environment.

9.6.1 Symbolic manipulation

Programming languages allowing us to perform symbolic computations, as well as numerical computations, should make our future research easier. This is reflected in the following sections.

9.6.2 Verification of existing equations

Only recently, have we started using Matlab [Mat12] to verify the correctness of the equations presented in this thesis. We are also investigating an alternative, Mathematica [Wel13]. We intend to use such tools to ensure that all existing equations are verified symbolically.

9.6.3 Generating new IFs

If the new symbolic manipulation tools prove up to the job, it would be nice to write suitable programs that generated new classes of IFs together with their asymptotic error constants automatically.

Appendix A

Equations and Convergence

This Appendix covers the material needed for generating the orders of convergence. It follows the order in which the IFs were introduced in Chapter 5. Firstly, a couple of definitions.

Let the IF $\phi(z)$ generate a sequence $z_0,\ z_1,\ \ldots,\ z_i,\ \ldots$ converging to α and let $\epsilon_i = z_i - \alpha$. If there exista a real number ρ and a non-zero constant C such that

$$\frac{|\epsilon_{i+1}|}{|\epsilon_i|^\rho} \to C \quad , \tag{A.1}$$

then ρ is called the *order* of the sequence and C is called the *asymptotic error constant*. Note that Equation (A.1) can be re-written as

$$\frac{|\phi(z) - \alpha|}{|z - \alpha|^\rho} \to C \quad . \tag{A.2}$$

An order is associated with an IF whether or not the generated sequence converges. The order assigned to an IF is the order of the generated sequence when it converges.

A.1 Simple Zeros

A.1.1 Equations for Simple Zeros

This section is concerned with deriving the order of convergence of our IFs. Therefore, we need expressions for $A_i(z), u(z)$, and their derivatives, evaluated

at α. We use the notation $\tilde{A}_i(z)$ to denote such an expansion, usually a Taylor series, of $A_i(z)$ about α. Similarly, for $\tilde{u}(z)$, $\tilde{T}_k(z)$, and so on.

Starting from first principles, we need the Taylor series expansion of $A_i(z)$ about α, namely

$$\tilde{A}_i(z) = A_i(\alpha) + A_i'(\alpha)\epsilon + \frac{A_i''(\alpha)}{2!}\epsilon^2 + \cdots \quad . \tag{A.3}$$

where $z = \alpha + \epsilon$ and the usual assumptions for the expansion of Equation (A.9) are assumed to hold (see [Hen74, pp. 142–143]). Differentiating Equation (2.7) on page 6, we obtain the following sequence of first-order derivatives.

$$\begin{aligned} A_2'(z) &= 3A_3(z) - 2A_2^2(z) \quad , \\ A_3'(z) &= 4A_4(z) - 2A_2(z)A_3(z) \quad , \\ A_4'(z) &= 5A_5(z) - 2A_2(z)A_4(z) \quad , \\ A_5'(z) &= 6A_6(z) - 2A_2(z)A_5(z) \quad , \end{aligned} \tag{A.4}$$

etc. Differentiating Equations (A.4), we obtain the following sequence of second-order derivatives.

$$\begin{aligned} A_2''(z) &= 3A_3'(z) - 4A_2(z)A_2'(z) \quad , \\ A_3''(z) &= 4A_4'(z) - 2A_2'(z)A_3(z) - 2A_2(z)A_3'(z) \quad , \\ A_4''(z) &= 5A_5'(z) - 2A_2'(z)A_4(z) - 2A_2(z)A_4'(z) \end{aligned} \tag{A.5}$$

etc. Differentiating Equations (A.5), we obtain the following sequence of third-order derivatives.

$$\begin{aligned} A_2'''(z) &= 3A_3''(z) - 4A_2'(z)^2 - 4A_2(z)A_2''(z) \quad , \\ A_3'''(z) &= 4A_4''(z) - 2A_2''(z)A_3(z) - 4A_2'(z)A_3'(z) - 2A_2(z)A_3''(z) \quad , \end{aligned} \tag{A.6}$$

etc. Differentiating Equations (A.6), we obtain the following fourth-order derivative

$$A_2^{(iv)}(z) = 3A_3'''(z) - 12A_2'(z)A_2''(z) - 4A_2(z)A_2'''(z) \quad . \tag{A.7}$$

Substituting Equation (A.4) through Equation (A.7) in Equation (A.3) on page 108

yields the following set of equations

$$\begin{aligned}
\tilde{A}_2(z) &= A_2(\alpha) - [2A_2^2(\alpha) - 3A_3(\alpha)]\epsilon \\
&\quad + [4A_2^3(\alpha) - 9A_2(\alpha)A_3(\alpha) + 6A_4(\alpha)]\epsilon^2 \\
&\quad - [8A_2^4(\alpha) - 24A_2^2(\alpha)A_3(\alpha) + 16A_2(\alpha)A_4(\alpha) + 9A_3^2(\alpha) \\
&\qquad - 10A_5(\alpha)]\epsilon^3 \quad , \\
\tilde{A}_3(z) &= A_3(\alpha) - [2A_2(\alpha)A_3(\alpha) - 4A_4(\alpha)]\epsilon \\
&\quad + [4A_2^2(\alpha)A_3(\alpha) - 8A_2(\alpha)A_4(\alpha) - 3A_3^2(\alpha) \\
&\qquad + 10A_5(\alpha)]\epsilon^2 \quad , \\
\tilde{A}_4(z) &= A_4(\alpha) - [2A_2(\alpha)A_4(\alpha) - 5A_5(\alpha)]\epsilon \quad ,
\end{aligned} \tag{A.8}$$

which will be required later. Note the use of $\tilde{\cdot}$ to denote an expansion about α.

Obviously even higher-order derivatives are required as we increase the order of convergence of this family of IFs.

Following the same procedure, for simple zeros, we need the Taylor series expansion of $u(z)$ about α, namely

$$u(z) = u(\alpha) + u'(\alpha)\epsilon + \frac{u''(\alpha)}{2!}\epsilon^2 + \cdots \quad . \tag{A.9}$$

Differentiating Equation (2.6) on page 6 and using Equation (2.7) on page 6, we derive the following equation.

$$u'(z) = 1 - 2A_2(z)u(z) \quad . \tag{A.10}$$

Differentiating Equation (A.10) repeatedly yields the following sequence of equations that we use later.

$$\begin{aligned}
u'(z) &= 1 - 2A_2(z)u(z) \quad , \\
u''(z) &= -2[A_2'(z)u(z) + A_2(z)u'(z)] \quad , \\
u'''(z) &= -2[A_2''(z)u(z) + 2A_2'(z)u'(z) + A_2(z)u''(z)] \quad , \\
u^{(iv)}(z) &= -2[A_2'''(z)u(z) + 3A_2''(z)u'(z) \\
&\quad + 3A_2'(z)u''(z) + A_2(z)u'''(z)] \quad , \\
u^{(v)}(z) &= -2[A_2^{(iv)}(z)u(z) + 4A_2'''(z)u'(z) + 6A_2''(z)u''(z) \\
&\quad + 4A_2'(z)u'''(z) + A_2(z)u^{(iv)}(z)] \quad ,
\end{aligned} \tag{A.11}$$

etc. Putting this together, after some elementary but heavy algebraic manipu-

lation, we have

$$\begin{aligned} u(\alpha) &= 0 \ , \\ u'(\alpha) &= 1 \ , \\ u''(\alpha) &= -2A_2(\alpha) \ , \\ u'''(\alpha) &= 12[A_2^2(\alpha) - A_3(\alpha)] \ , \\ u^{(iv)}(\alpha) &= -24[4A_2^3(\alpha) - 7A_2(\alpha)A_3(\alpha) + 3A_4(\alpha)] \ , \\ u^{(v)}(\alpha) &= 240[4A_2^4(\alpha) - 10A_2^2(\alpha)A_3(\alpha) + 3A_3^2(\alpha) \\ &\qquad + 5A_2(\alpha)A_4(\alpha) - 2A_5(\alpha)] \ , \end{aligned} \tag{A.12}$$

so that Equation (A.9) on page 109 can be re-written as

$$\begin{aligned} \tilde{u}(z) = \epsilon &- A_2(\alpha)\epsilon^2 + 2[A_2^2(\alpha) - A_3(\alpha)]\epsilon^3 \\ &- [4A_2^3(\alpha) - 7A_2(\alpha)A_3(\alpha) + 3A_4(\alpha)]\epsilon^4 \\ &+ 2[4A_2^4(\alpha) - 10A_2^2(\alpha)A_3(\alpha) + 5A_2(\alpha)A_4(\alpha) \\ &\qquad + 3A_3^2(\alpha) - 2A_5(\alpha)]\epsilon^5 + \cdots \end{aligned} \ . \tag{A.13}$$

Both Equation (A.8) and Equation (A.13) are used in deriving the order of convergence of our various IFs for simple zeros.

A.1.2 Convergence of One-point IFs for Simple Zeros

Isaac Newton's second-order IF

Given Equation (5.13) on page 50 and Equation (A.13) on page 110 we obtain the order of convergence as follows

$$\begin{aligned} \hat{\epsilon}_\nu &= \epsilon_\nu - \tilde{u}(z_\nu) \ , \\ &= \epsilon_\nu - \left[\epsilon_\nu - A_2(\alpha_\nu)\epsilon_\nu^2\right] + O(\epsilon_\nu^3) \ , \\ &= A_2(\alpha_\nu)\epsilon_\nu^2 + O(\epsilon_\nu^3) \ , \end{aligned} \tag{A.14}$$

which is the required result.

Edmond Halley's third-order IF

Given Equation (5.14) on page 50 and Equation (A.13) on page 110 we

obtain the order of convergence as follows

$$\begin{aligned}\hat{\epsilon}_\nu &= \epsilon_\nu - \tilde{u}(z_\nu) - \tilde{A}_2(z_\nu)\tilde{u}^2(z_\nu) \quad , \\ &= \epsilon_\nu - \left\{\epsilon_\nu - A_2(\alpha_\nu)\epsilon_\nu^2 + 2\left[A_2^2(\alpha_\nu) - A_3(\alpha_\nu)\right]\epsilon_\nu^3\right\} \\ &\qquad - \left\{A_2(\alpha_\nu) - \left[2A_2(\alpha_\nu) - 3A_3(\alpha_\nu)\right]\epsilon_\nu\right\} \\ &\qquad\quad \left[\epsilon_\nu^2 - 2A_2(\alpha_\nu)\epsilon_\nu^3\right] + O(\epsilon_\nu^4) \quad , \\ &= \left[2A_2^2(\alpha_\nu) - A_3(\alpha_\nu)\right]\epsilon^3 + O(\epsilon_\nu^4) \quad ,\end{aligned} \tag{A.15}$$

which is the required result.

I Kiss' fourth-order IF

Given Equation (5.15) on page 51 we obtain the order of convergence as follows

$$\begin{aligned}\hat{\epsilon}_\nu &= \epsilon_\nu - \tilde{u}(z_\nu) - \tilde{A}_2(z_\nu)\tilde{u}^2(z_\nu) - \left[2\tilde{A}_2^2(z_\nu) - \tilde{A}_3(z_\nu)\right]\tilde{u}^3(z_\nu) \quad , \\ &= \ldots \quad , \\ &= \left[5A_2^3(\alpha_\nu) - 5A_2(\alpha_\nu)A_3(\alpha_\nu) + A_4(\alpha_\nu)\right]\epsilon_\nu^4 + O(\epsilon_\nu^5) \quad ,\end{aligned} \tag{A.16}$$

which is the required result.

I Kiss' fifth-order IF

Given Equation (5.16) on page 51 we obtain the order of convergence as follows

$$\begin{aligned}\hat{\epsilon}_\nu &= \epsilon_\nu - \tilde{u}(z_\nu) - \tilde{A}_2(z_\nu)\tilde{u}^2(z_\nu) - \left[2\tilde{A}_2^2(z_\nu) - \tilde{A}_3(z_\nu)\right]\tilde{u}^3(z_\nu) \\ &\qquad - \left[5\tilde{A}_2^3(z_\nu) - 5\tilde{A}_2(z_\nu)\tilde{A}_3(z_\nu) + \tilde{A}_4(z_\nu)\right]\tilde{u}^4(z_\nu) \quad , \\ &= \ldots \quad , \\ &= \left[14A_2^4(\alpha_\nu) - 21A_2^2(\alpha_\nu)A_3(\alpha_\nu) + 6A_2(\alpha_\nu)A_4(\alpha_\nu)\right. \\ &\qquad\qquad \left. + 3A_3^2(\alpha_\nu) - A_5(\alpha_\nu)\right]\epsilon_\nu^5 + O(\epsilon_\nu^6) \quad ,\end{aligned} \tag{A.17}$$

which is the required result.

Note that Equation (A.14) and Equation (A.15) were verified using a Matlab program, see §C.6.2 on page 200. Equation (A.16) and Equation (A.17) were generated by the same Matlab program.

A.1.3 Convergence of Simultaneous IFs for Simple Zeros

For those IFs involving simultaneous approximations, we also need the expansion for $T_k(z_\nu)$. Only a first-order approximation is required. Note this is for

simple zeros.

$$
\begin{aligned}
T_k(z_\nu) &= \sum_{i\neq\nu} \frac{1}{(z_\nu - z_i)^k} \quad , \\
&= \sum_{i\neq\nu} \frac{1}{(\alpha_\nu - \alpha_i)^k} \left[1 + \frac{\epsilon_\nu - \epsilon_i}{\alpha_\nu - \alpha_i}\right]^{-k} \quad , \\
&= \sum_{i\neq\nu} \frac{1}{(\alpha_\nu - \alpha_i)^k} - k \sum_{i\neq\nu} \frac{\epsilon_\nu - \epsilon_i}{(\alpha_\nu - \alpha_i)^{k+1}} + O(\epsilon^2) \quad , \\
&= \sum_{i\neq\nu} \frac{1}{(\alpha_\nu - \alpha_i)^k} - k\epsilon_\nu \sum_{i\neq\nu} \frac{1}{(\alpha_\nu - \alpha_i)^{k+1}} \\
&\qquad\qquad + k \sum_{i\neq\nu} \frac{\epsilon_i}{(\alpha_\nu - \alpha_i)^{k+1}} + O(\epsilon^2) \quad , \\
&= S_k(\alpha_\nu) - kS_{k+1}(\alpha_\nu)\epsilon_\nu + k \sum_{i\neq\nu} \frac{\epsilon_i}{(\alpha_\nu - \alpha_i)^{k+1}} + O(\epsilon^2) \quad .
\end{aligned}
\tag{A.18}
$$

Given that

$$
\begin{aligned}
S_1(\alpha_\nu) &= A_2(\alpha_\nu) \quad , \\
S_2(\alpha_\nu) &= A_2^2(\alpha_\nu) - 2A_3(\alpha_\nu) \quad , \\
S_3(\alpha_\nu) &= A_2^3(\alpha_\nu) - 3A_2(\alpha_\nu)A_3(\alpha_\nu) + 3A_4(\alpha_\nu) \quad , \\
S_4(\alpha_\nu) &= A_2^4(\alpha_\nu) - 4A_2^3(\alpha_\nu)A_3(\alpha_\nu) + 2A_3^2(\alpha_\nu) + 4A_2(\alpha_\nu)A_4(\alpha_\nu) \\
&\qquad - 4A_5(\alpha_\nu) \quad ,
\end{aligned}
\tag{A.19}
$$

we finally have

$$
\begin{aligned}
\tilde{T}_1(z_\nu) &= A_2(\alpha_\nu) - \left[3A_2^2(\alpha_\nu) - 5A_3(\alpha_\nu)\right]\epsilon_\nu \\
&\qquad + \sum_{i\neq\nu} \frac{\epsilon_i}{(\alpha_\nu - \alpha_i)^2} + O(\epsilon^2) \quad , \\
\tilde{T}_2(z_\nu) &= A_2^2(\alpha_\nu) - 2A_3(\alpha_\nu) \\
&\qquad - 2\left[3A_2^3(\alpha_\nu) - 8A_2(\alpha_\nu)A_3(\alpha_\nu) + 7A_4(\alpha_\nu)\right]\epsilon_\nu \\
&\qquad + 2\sum_{i\neq\nu} \frac{\epsilon_i}{(\alpha_\nu - \alpha_i)^3} + O(\epsilon^2) \quad , \\
\tilde{T}_3(z_\nu) &= A_2^3(\alpha_\nu) - 3A_2(\alpha_\nu)A_3(\alpha_\nu) + 3A_4(\alpha_\nu) \\
&\qquad - 3\left[A_2^4(\alpha_\nu) - 4A_2^3(\alpha_\nu)A_3(\alpha_\nu) + 10A_2(\alpha_\nu)A_4(\alpha_\nu)\right. \\
&\qquad\qquad \left. + 5A_3^2(\alpha_\nu) - 9A_5(\alpha_\nu)\right] \\
&\qquad + 3\sum_{i\neq\nu} \frac{\epsilon_i}{(\alpha_\nu - \alpha_i)^4} + O(\epsilon^2) \quad .
\end{aligned}
\tag{A.20}
$$

We need powers of $u(z_\nu)$ in order to verify the IF's orders of convergence and their asymptotic error constants. We truncate all equations at ϵ_ν^5 as this is the highest order IFs we discuss.

$$\begin{aligned}
\tilde{u}(z_\nu) &= \epsilon_\nu - A_2(\alpha_\nu)\epsilon_\nu^2 + 2[A_2^2(\alpha_\nu) - A_3(\alpha_\nu)]\epsilon_\nu^3 \\
&\quad - [4A_2^3(\alpha_\nu) - 7A_2(\alpha_\nu)A_3(\alpha_\nu) + 3A_4(\alpha_\nu)]\epsilon_\nu^4 \\
&\quad + 2[4A_2^4(\alpha_\nu) - 10A_2^2(\alpha_\nu)A_3(\alpha_\nu) \\
&\qquad\qquad + 5A_2(\alpha_\nu)A_4(\alpha_\nu) \\
&\qquad\qquad + 3A_3^2(\alpha_\nu) - 2A_5(\alpha_\nu)]\epsilon_\nu^5 + \cdots \quad , \\
\tilde{u}^2(z_\nu) &= \epsilon_\nu^2 - 2A_2(\alpha_\nu)\epsilon_\nu^3 + [5A_2^2(\alpha_\nu) - 4A_3(\alpha_\nu)]\epsilon_\nu^4 \\
&\quad - [12A_2^3(\alpha_\nu) - 18A_2(\alpha_\nu)A_3(\alpha_\nu) + 6A_4(\alpha_\nu)]\epsilon_\nu^5 \quad , \\
\tilde{u}^3(z_\nu) &= \epsilon_\nu^3 - 3A_2(\alpha_\nu)\epsilon_\nu^4 + [9A_2^2(\alpha_\nu) - 6A_3(\alpha_\nu)]\epsilon_\nu^5 \\
\tilde{u}^4(z_\nu) &= \epsilon_\nu^4 - 4A_2(\alpha_\nu)\epsilon_\nu^5 \quad , \\
\tilde{u}^5(z_\nu) &= \epsilon_\nu^5 \quad .
\end{aligned} \tag{A.21}$$

Second-order IF

There is no polynomial form for this second-order IF.

Mick Farmer and George Loizou's third-order IF

Given Equation (5.24) on page 53 and Equation (A.20) on page 112 we obtain the order of convergence as follows

$$\begin{aligned}
\hat{\epsilon}_\nu &= \epsilon_\nu - \tilde{u}(z_\nu) - \tilde{T}_1(z_\nu)\tilde{u}^2(z_\nu) \quad , \\
&= \epsilon_\nu - \{\epsilon_\nu - A_2(\alpha_\nu)\epsilon_\nu^2 + 2[A_2^2(\alpha_\nu) - A_3(\alpha_\nu)]\epsilon_\nu^3\} \\
&\qquad - \left\{ A_2(\alpha_\nu) - [3A_2^2(\alpha_\nu) - 5A_3(\alpha_\nu)]\epsilon_\nu + \sum_{i\neq\nu} \frac{\epsilon_i}{(\alpha_\nu - \alpha_i)^2} \right\} \\
&\qquad\quad [\epsilon_\nu^2 - 2A_2(\alpha_\nu)\epsilon_\nu^3] \quad , \\
&= 3[A_2^2(\alpha_\nu) - A_3(\alpha_\nu)]\epsilon_\nu^3 - \epsilon_\nu^2 \sum_{i\neq\nu} \frac{\epsilon_i}{(\alpha_\nu - \alpha_i)^2} \quad ,
\end{aligned} \tag{A.22}$$

which is the required result.

Mick Farmer and George Loizou's fourth-order IF

Given Equation (5.25) on page 53 we obtain the order of convergence as follows

$$\begin{aligned}
\hat{\epsilon}_\nu &= \epsilon_\nu - \tilde{u}(z_\nu) - \tilde{A}_2(z_\nu)\tilde{u}^2(z_\nu) - \frac{1}{2}\left[3\tilde{A}_2^2(z_\nu) + \tilde{T}_2(z_\nu)\right]\tilde{u}^3(z_\nu) \quad , \\
&= \epsilon_\nu - A_2(\alpha_\nu)\epsilon_\nu^2 + 2\left[A_2^2(\alpha_\nu) - A_3(\alpha_\nu)\right]\epsilon_\nu^3 \\
&\quad - \left[4A_2^3(\alpha_\nu) - 7A_2(\alpha_\nu)A_3(\alpha_\nu) + 3A_4(\alpha_\nu)\right]\epsilon_\nu^4 \\
&\quad - \left\{A_2(\alpha_\nu) - \left[2A_2^2(\alpha_\nu) - 3A_3(\alpha_\nu)\right]\epsilon_\nu\right. \\
&\qquad \left. + \left[4A_2^3(\alpha_\nu) - 9A_2(\alpha_\nu)A_3(\alpha_\nu) + 6A_4(\alpha_\nu)\right]\epsilon_\nu^2\right\} \\
&\qquad \left\{\epsilon_\nu^2 - 2A_2(\alpha_\nu)\epsilon_\nu^3 + \left[5A_2^2(\alpha_\nu) - 4A_3(\alpha_\nu)\right]\epsilon_\nu^4\right\} \\
&\qquad - \frac{1}{2}\left[3A_2^2(\alpha_\nu) + \sum_{i\neq\nu}\frac{\epsilon_i}{(\alpha_\nu - \alpha_i)^3}\right] \\
&\qquad \left[\epsilon_\nu^3 - 3A_2(\alpha_\nu)\epsilon_\nu^4\right] \quad , \\
&= 2\left[3A_2^3(\alpha_\nu) - 4A_2(\alpha_\nu)A_3(\alpha_\nu) + 2A_4(\alpha_\nu)\right]\epsilon_\nu^4 \\
&\quad + \epsilon_\nu^3\sum_{i\neq\nu}\frac{\epsilon_i}{(\alpha_\nu - \alpha_i)^3} \quad ,
\end{aligned} \tag{A.23}$$

which is the required result.

Mick Farmer and George Loizou's fifth-order IF

$$\begin{aligned}
\hat{\epsilon}_\nu &= \epsilon_\nu - \tilde{u}(z_\nu) - \tilde{A}_2(z_\nu)\tilde{u}^2(z_\nu) \\
&\quad - \left[2\tilde{A}_2^2(z_\nu) - \tilde{A}_3(z_\nu)\right]\tilde{u}^3(z_\nu) \\
&\quad - \frac{1}{3}\left[14\tilde{A}_2^3(z_\nu) + 12\tilde{A}_2(z_\nu)\tilde{A}_3(z_\nu) + \tilde{T}_3(z_\nu)\right]\tilde{u}^4(z_\nu) \\
&= \ldots \quad , \\
&= \left[15A_2^4(\alpha_\nu) - 4A_2^3(\alpha_\nu)A_3(\alpha_\nu) + 21A_2^2(\alpha_\nu)A_3(\alpha_\nu)\right. \\
&\qquad \left. + 10A_2(\alpha_\nu)A_4(\alpha_\nu) + 5A_3^2(\alpha_\nu) - 5A_5(\alpha_\nu)\right]\epsilon_\nu^5 \\
&\quad - \epsilon_\nu^4\sum_{i\neq\nu}\frac{\epsilon_i}{(\alpha_\nu - \alpha_i)^4} \quad ,
\end{aligned} \tag{A.24}$$

which is the required result.

A.1.4 Convergence of Variable-order IFs for Simple Zeros

In this thesis we have introduced only one example of this class of IF, namely Equation (5.28) on page 55, which is based on our third-order simultaneous IF, given in Equation (5.24) on page 53. We accept that it is really a multipoint

IF and it is an area of our current research. See §9.2 for more details of our research in this area.

A.2 Multiple Zeros

A.2.1 Equations for Multiple Zeros

The convergence of the IFs for multiple zeros involves, once again, using sufficient terms in the Taylor series expansion of $u(z)$ about $u(\alpha)$, namely Equation (A.9) on page 109. The following definition is a slight variation on Joseph Traub's $B_{j,m}$ which he calls the *generalised normalised Taylor series coefficient* [Tra64, p. 6],

$$C_i(z_\nu) = \frac{1}{m_\nu} \frac{p^{(m_\nu+i-1)}(z_\nu)}{(m_\nu+i-1)!} \frac{(m_\nu)!}{p^{(m_\nu)}(z_\nu)} \quad , \; i = 1,\; 2,\; \ldots \quad , \tag{A.25}$$

which is used when the multiplicity $m_\nu > 1$. Note that when $m_\nu = 1$, then

$$A_i(z_\nu) \equiv B_i(z_\nu) \equiv C_i(z_\nu) \quad . \tag{A.26}$$

In the case of a multiple zero we have

$$\begin{aligned} p(z_\nu) &= \sum_{i=m_\nu}^{\infty} \frac{p^{(i)}(\alpha_\nu)}{i!} \epsilon^i \quad , \\ p'(z_\nu) &= \sum_{i=m_\nu}^{\infty} i \frac{p^{(i)}(\alpha_\nu)}{i!} \epsilon^{i-1} \quad . \end{aligned} \tag{A.27}$$

Given Equation (2.6) on page 6, dividing the expansion of $p(z_\nu)$ by the expansion of $p'(z_\nu)$, both from Equation (A.27), yields the following coefficients of $m_\nu u(z_\nu)$

$$\begin{aligned} u(\alpha_\nu) &= 0 \quad , \\ u'(\alpha_\nu) &= 1 \quad , \\ u''(\alpha_\nu) &= -C_2(\alpha_\nu) \quad , \\ u'''(\alpha_\nu) &= (m_\nu+1)C_2^2(\alpha_\nu) - 2C_3(\alpha_\nu) \quad , \\ u^{(iv)}(\alpha_\nu) &= -(m_\nu+1)^2 C_2^3(\alpha_\nu) + (3m_\nu+4)C_2(\alpha_\nu)C_3(\alpha_\nu) \\ &\qquad - 3C_4(\alpha_\nu) \quad , \end{aligned} \tag{A.28}$$

so that Equation (A.9) on page 109 can be re-written *for multiple zeros* as

$$\begin{aligned}\tilde{u}(z_\nu) = \frac{1}{m_\nu}\big\{&\epsilon_\nu - C_2(\alpha_\nu)\epsilon_\nu^2 + \big[(m_\nu+1)C_2^2(\alpha_\nu) - 2C_3(\alpha_\nu)\big]\,\epsilon_\nu^3 \\ &- \big[(m_\nu+1)^2 C_2^3(\alpha_\nu) + (3m_\nu+4)C_2(\alpha_\nu)C_3(\alpha_\nu) \\ &\qquad -3C_4(\alpha_\nu)\big]\,\epsilon_\nu^4 \\ &+ \big[(m_\nu+1)^3 C_2^4(\alpha_\nu) \\ &\quad -2(m_\nu+1)(2m_\nu+3)C_2^2(\alpha_\nu)C_3(\alpha_\nu) \\ &\quad +2(2m_\nu+3)C_2(\alpha_\nu)C_4(\alpha_\nu) + 2(m_\nu+2)C_3^2(\alpha_\nu) \\ &\quad -4C_5(\alpha_\nu)\big]\,\epsilon^5 + \ldots\big\} \quad .\end{aligned} \tag{A.29}$$

Note that this time the notation $\tilde{f}(z_\nu)$ stands for the Taylor series expansion of $f(z_\nu)$ about α_ν, an explicit approximation.

A.2.2 Convergence of One-point IFs for Multiple Zeros

Louis Rall's second-order modified Newton IF

From Equation (5.36) on page 57 we have

$$\begin{aligned}\hat{\epsilon}_\nu &= \epsilon_\nu - m_\nu\tilde{u}(z_\nu) \quad , \\ &= \epsilon_\nu - m_\nu\left[\frac{\epsilon_\nu}{m_\nu} - \frac{C_2(\alpha_\nu)}{m_\nu}\epsilon_\nu^2 + O(\epsilon_\nu^3)\right] \quad , \\ &= C_2(\alpha_\nu)\epsilon_\nu^2 + O(\epsilon_\nu^3) \quad ,\end{aligned} \tag{A.30}$$

which is the required result.

Joseph Traub's third-order IF

From Equation (5.37) on page 58 we have

$$\begin{aligned}
\hat{\epsilon}_\nu &= \epsilon_\nu + m_\nu \left(\frac{m_\nu - 3}{2}\right) \tilde{u}(z_\nu) - m_\nu^2 \tilde{A}_2(z_\nu)\tilde{u}^2(z_\nu) \quad , \\
&= \epsilon_\nu + m_\nu \left(\frac{m_\nu - 3}{2}\right) \left\{ \frac{\epsilon_\nu}{m_\nu} - \frac{C_2(\alpha_\nu)}{m_\nu}\epsilon_\nu^2 \right. \\
&\qquad \left. + \left[\frac{(m_\nu + 1)C_2^2(\alpha_\nu) - 2C_3(\alpha_\nu)}{m_\nu}\right] \epsilon_\nu^3 \right\} \\
&\quad - m_\nu^2 \left\{ \left(\frac{m_\nu - 1}{2\epsilon_\nu}\right) + (m_\nu + 1)C_2(\alpha_\nu) \right. \\
&\qquad - \left[\left(\frac{m_\nu^2 + 2m_\nu + 1}{2}\right) C_2^2(\alpha_\nu) \right. \\
&\qquad \left. \left. - \ (m_\nu - 2)C_3(\alpha_\nu)\right] \epsilon_\nu \right\} \\
&\quad \left\{ \frac{\epsilon_\nu^2}{m_\nu^2} - 2\frac{C_2(\alpha_\nu)}{m_\nu^2}\epsilon_\nu^3 \right. \\
&\qquad \left. + \left[\frac{(2m_\nu + 3)C_2^2(\alpha_\nu) - 4C_3(\alpha_\nu)}{m_\nu^2}\epsilon_\nu^4\right] \right\} \quad , \\
&= \dots \quad , \\
&= \left[\left(\frac{m_\nu + 3}{2}\right) C_2^2(\alpha_\nu) - C_3(\alpha_\nu)\right] \epsilon_\nu^3 + O(\epsilon_\nu^4) \quad ,
\end{aligned} \tag{A.31}$$

which is the required result.

Joseph Traub's fourth-order IF

From Equation (5.38) on page 58 we have

$$\begin{aligned}
\hat{\epsilon}_\nu &= \epsilon_\nu - m_\nu \left(\frac{m_\nu^2 - 6m_\nu + 11}{6}\right) \tilde{u}(z_\nu) \\
&\qquad + m_\nu^2 (m_\nu - 2)\tilde{A}_2(z_\nu)\tilde{u}^2(z_\nu) \\
&\qquad - m_\nu^3 \left[2\tilde{A}_2^2(z_\nu) - \tilde{A}_3(z_\nu)\right] \tilde{u}^3(z_\nu) \\
&= \dots \quad , \\
&= \left[\left(\frac{m_\nu^2 + 6m_\nu + 8}{3}\right) C_2^3(\alpha_\nu) - (m_\nu + 4)C_2(\alpha_\nu)C_3(\alpha_\nu) \right. \\
&\qquad \left. + C_4(\alpha_\nu)\right] \epsilon_\nu^4 + O(\epsilon_\nu^5) \quad ,
\end{aligned} \tag{A.32}$$

which is the required result.

Joseph Traub's fifth-order IF

From Equation (5.39) on page 58 we have

$$\begin{aligned}
\hat{\epsilon}_\nu &= \epsilon_\nu + m_\nu \left(\frac{m_\nu^3 - 10m_\nu^2 + 35m_\nu - 50}{24} \right) \tilde{u}(z_\nu) \\
&\quad - m_\nu^2 \left(\frac{7m_\nu^2 - 30m_\nu + 35}{12} \right) \tilde{A}_2(z_\nu) \tilde{u}^2(z_\nu) \\
&\quad + m_\nu^3 \left(\frac{3m_\nu - 5}{2} \right) \left[2\tilde{A}_2^2(z_\nu) - \tilde{A}_3(z_\nu) \right] \tilde{u}^3(z_\nu) \\
&\quad - m_\nu^4 \left[5\tilde{A}_2^3(z_\nu) - 5\tilde{A}_2(z_\nu)\tilde{A}_3(z_\nu) + \tilde{A}_4(z_\nu) \right] \tilde{u}^4(z_\nu) \quad , \\
&= \dots \quad , \\
&= \left[\left(\frac{6m_\nu^3 + 55m_\nu^2 + 150m_\nu + 125}{24} \right) C_2^4(\alpha_\nu) \right. \\
&\quad - \left(\frac{2m_\nu^2 + 15m_\nu + 25}{2} \right) C_2^2(\alpha_\nu) C_3(\alpha_\nu) \\
&\quad + (m_\nu + 5)\, C_2(\alpha_\nu) C_4(\alpha_\nu) \\
&\quad \left. + \left(\frac{m_\nu + 5}{2} \right) C_3^2(\alpha_\nu) - C_5(\alpha_\nu) \right] \epsilon_\nu^5 + O(\epsilon_\nu^6) \quad ,
\end{aligned} \tag{A.33}$$

which is, we hope, the required result.

A.2.3 Convergence of Simultaneous IFs for Multiple Zeros

Following the same approach as used for simple zeros in Equation (A.18) on page 112, we have for *multiple* zeros

$$\begin{aligned}
T_k(z_\nu) &= \sum_{i \neq \nu} \frac{m_i}{(z_\nu - z_i)^k} \quad , \\
&= \sum_{i \neq \nu} \frac{m_i}{(\alpha_\nu - \alpha_i)^k} \left[1 + \frac{\epsilon_\nu - \epsilon_i}{\alpha_\nu - \alpha_i} \right]^{-k} \quad , \\
&= \dots \quad , \\
&= S_k(\alpha_\nu) - k S_{k+1}(\alpha_\nu)\epsilon_\nu + k \sum_{i \neq \nu} \frac{m_i \epsilon_i}{(\alpha_\nu - \alpha_i)^{k+1}} + O(\epsilon^2) \quad .
\end{aligned} \tag{A.34}$$

Second-order IF

There is no second-order simultaneous IF in polynomial form.

Mick Farmer and George Loizou's third-order IF

$$\begin{aligned}\hat{\epsilon}_\nu &= \epsilon_\nu - m_\nu \tilde{u}(z_\nu) - m_\nu \tilde{T}_1(z_\nu)\tilde{u}^2(z_\nu) \quad , \\ &= \dots \quad , \\ &= 2[C_2^2(\alpha_\nu) - C_3(\alpha_\nu)] - \frac{1}{m}\epsilon_\nu^2 \sum_{i \neq \nu} \frac{\epsilon_\nu}{(\alpha_\nu - \alpha_i)^2} \quad ,\end{aligned} \tag{A.35}$$

which is the required result.

Mick Farmer and George Loizou's fourth-order IF

Asymptotic error constant has not been established.

Mick Farmer and George Loizou's fifth-order IF

Asymptotic error constant has not been established.

A.2.4 Convergence of Variable-order IFs for Multiple Zeros

These variable-order IFs are still new to us. As we said in §A.1.4 on page 114, this is a topic for our further research.

Appendix B

IFs in Rational Form

Many of the IFs presented in this thesis were originally derived in their rational form. Although we prefer to present these IFs in their polynomial form, it is useful to maintain links to those original derivations. That is the intention of this Appendix. Cross-references are provided between both forms in all cases.

However, it is important to understand that there may be a number of different rational forms of the same order, depending on the degrees of the numerator and denominator. Joseph Traub has investigated these [Tra64, pp. 88–92] where he uses a **Padé** table to collect them together. He suggests, via a couple of references, that those IFs that lie near the diagonal of the Padé table are best. This topic will not be covered any further.

B.1 Simple IFs in Rational Form

B.1.1 One-point IFs

The first four equations are the rational forms corresponging to the one-point IFs for simple zeros described in §5.3.1 starting on page 50.

Isaac Newton's second-order IF [New36]

$$\hat{z}_\nu = z_\nu - \frac{p(z_\nu)}{p'(z_\nu)} \quad . \tag{B.1}$$

Edmond Halley's third-order IF [Hal94]

$$\hat{z}_\nu = z_\nu - \frac{u(z_\nu)}{1 - A_2(z_\nu)u(z_\nu)} \quad , \tag{B.2}$$

which is often referred to as Chebyshev's method and stated by Peter Jarratt in [Jar69, p. 121] as *well-known*. According to Joseph Traub [Tra64, p. 91], this IF has the distinction of being the most frequently rediscovered IF in the literature. It was also derived two centuries later by Ernst Schröder [Sch70, p. 352].

I Kiss' fourth-order IF [Kis54, p. 68]

$$\hat{z}_\nu = z_\nu - \frac{u(z_\nu)[1 - A_2(z_\nu)u(z_\nu)]}{1 - 2A_2(z_\nu)u(z_\nu) + A_3(z_\nu)u^2(z_\nu)} \quad . \tag{B.3}$$

I Kiss' fifth-order [Kis54, p. 68]

$$\hat{z}_\nu = z_\nu - \frac{u(z_\nu)[1 - 2A_2(z_\nu) + A_3(z_\nu)u^2(z_\nu)]}{\begin{aligned} &1 - 3A_2(z_\nu)u(z_\nu) \\ &\quad + [2A_3(z_\nu) + A_2^2(z_\nu)]u^2(z_\nu) \\ &\quad - A_4(z_\nu)u^3(z_\nu) \end{aligned}} \quad . \tag{B.4}$$

B.1.2 Simultaneous IFs

The next four equations are the rational forms corresponding to the simultaneous IFs for simple zeros described in §5.3.2 starting on page 51.

Kiril Dochev and Byrnev's second-order IF [DB64, p. 174]

$$\hat{z}_\nu = z_\nu - \frac{p(z_\nu)}{\prod_{i \neq \nu}(z_\nu - z_i)} \quad . \tag{B.5}$$

Since its original introduction in 1964, this IF has been rediscovered many times [Ker66, Ehr67, Pre71, GH72], especially when the polynomial is **monic**, i.e. $a_n = 1$. Interestingly, the term after the first minus sign (-) is called *Weierstrass's correction* in [PPŽ03, p. 2].

Louis Ehrlich's third-order IF [Ehr67, p. 107]

$$\hat{z}_\nu = z_\nu - \frac{u(z_\nu)}{1 - T_1(z_\nu)u(z_\nu)} \quad . \tag{B.6}$$

Many researchers refer to this as *Aberth's Method* [1], but that paper was published much later in 1973 [Abe73].

[1] Occasionally as the Aberth-Ehrlich, or Ehrlich-Aberth, Method.

Mick Farmer and George Loizou's fourth-order IF [FL75, p. 252]

$$\hat{z}_\nu = z_\nu - \frac{u(z_\nu)[1 - A_2(z_\nu)u(z_\nu)]}{1 - 2A_2(z_\nu)u(z_\nu) + \frac{1}{2}[A_2^2(z_\nu) - T_2(z_\nu)]u^2(z_\nu)} \tag{B.7}$$

Mick Farmer and George Loizou's fifth-order IF [FL75, p. 252]

$$\hat{z}_\nu = z_\nu - \frac{u(z_\nu)[1 - A_2(z_\nu)u(z_\nu) + A_3(z_\nu)u^2(z_\nu)]}{\begin{aligned} &1 - 3A_2(z_\nu)u(z_\nu) \\ &+ [2A_3(z_\nu) + A_2^2(z_\nu)]u^2(z_\nu) \\ &- \frac{1}{3}[3A_2(z_\nu)A_3(z_\nu) - A_2^3(z_\nu) + T_3(z_\nu)]u^3(z_\nu) \end{aligned}} \quad . \tag{B.8}$$

B.2 Multiple IFs in Rational Form

B.2.1 One-point IFs

The first four equations are the rational forms of the one-point IFs for multiple zeros described in §5.4.1 starting on page 57.

Louis Rall's second-order modified Newton IF [Ral66]

$$\hat{z}_\nu = z_\nu - \left[\frac{p(z_\nu)}{\prod_{i \neq \nu}(z_\nu - z_i)^{m_i}}\right]^{\frac{1}{m_\nu}} \quad . \tag{B.9}$$

Hansen-Patrick family (third-order) IF [HP77, p. 265]

$$\hat{z}_\nu = z_\nu - \frac{m_\nu u(z_\nu)}{\frac{m_\nu + 1}{2!} - m_\nu A_2(z_\nu)u(z_\nu)} \quad . \tag{B.10}$$

Joseph Traub's fourth-order IF [Tra64, p. 139]

$$\hat{z}_\nu = z_\nu - \frac{m_\nu \left[\frac{m_\nu + 1}{2!} - m_\nu A_2(z_\nu)u(z_\nu)\right] u(z_\nu)}{\begin{aligned} &\frac{(m_\nu + 1)(2m_\nu + 1)}{3!} - m_\nu(m_\nu + 1)A_2(z_\nu)u(z_\nu) \\ &+ m_\nu^2 A_3(z_\nu)u^2(z_\nu) \end{aligned}} \quad , \tag{B.11}$$

which we defined in [FL77, p. 429].

Joseph Traub's fifth-order IF [Tra64, p. 139]

$$\hat{z}_\nu = z_\nu - \frac{m_\nu \left[\begin{array}{l} \dfrac{(m_\nu+1)(2m_\nu+1)}{3!} \\ -\, m_\nu(m_\nu+1)A_2(z_\nu)u(z_\nu) \\ +\, m_\nu^2 A_3(z_\nu)u^2(z_\nu) \end{array} \right] u(z_\nu)}{\begin{array}{l} \dfrac{(m_\nu+1)(2m_\nu+1)(3m_\nu+1)}{4!} \\ -\, m_\nu \dfrac{(m_\nu+1)(2m_\nu+1)}{2!} A_2(z_\nu)u(z_\nu) \\ +\, m_\nu^2(m_\nu+1)\left[\dfrac{A_2^2(z_\nu)}{2} + A_3(z_\nu)\right] u^2(z_\nu) \\ -\, m_\nu^3 A_4(z_\nu)u^3(z_\nu) \end{array}} \quad , \tag{B.12}$$

which we defined in [FL77, p. 429].

B.2.2 Simultaneous IFs

The next four equations are the rational forms of the simultaneous IFs for multiple zeros described in §5.4.2 starting on page 58.

Second-order IF

$$\hat{z}_\nu = z_\nu - \left[\frac{p(z_\nu)}{\prod_{i \neq \nu} (z_\nu - z_i)^{m_i}} \right]^{\frac{1}{m_\nu}} \quad . \tag{B.13}$$

This IF cannot be derived as outlined above[2], but comes from a direct expansion of $p(z_\nu)$, namely

$$p(z_\nu) = (z_\nu - \alpha_\nu)^{m_\nu} \prod_{i \neq \nu} (z_\nu - z_i)^{m_i} + O(\epsilon^{m_\nu+1}) \quad . \tag{B.14}$$

Louis Ehrlich's third-order IF [Ehr67]

$$\hat{z}_\nu = z_\nu - \frac{m_\nu u(z_\nu)}{1 - T_1(z_\nu)u(z_\nu)} \quad . \tag{B.15}$$

A paper by Abdel Anourein [Ano77, pp. 244–245] introduces an improvement to Equation (B.15) to fourth-order similar to our variable-order IF, see Equation (5.28) on page 55.

Mick Farmer and George Loizou's fourth-order IF [FL77, p. 430]

[2] which would yield Rall's modified Newton approximation, Equation (5.36)

$$\hat{z}_\nu = z_\nu - \frac{m_\nu \left[\frac{1}{2}(m_\nu+1) - m_\nu A_2(z_\nu)u(z_\nu)\right] u(z_\nu)}{\begin{array}{rl} & - \frac{m_\nu}{2}(m_\nu+3)A_2(z_\nu)u(z_\nu) \\ \frac{1}{8}(m_\nu^2+6m_\nu+1) & + \frac{m_\nu^2}{2}A_2^2(z_\nu)u^2(z_\nu) \\ & - \frac{m_\nu}{2}T_2(z_\nu)u^2(z_\nu) \end{array}} \quad . \tag{B.16}$$

Mick Farmer and George Loizou's fifth-order IF [FL77, p. 430]

$$\hat{z}_\nu = z_\nu - \frac{m_\nu \left[\begin{array}{l} \frac{1}{3!}(m_\nu+1)(2m_\nu+1) \\ - m_\nu(m_\nu+1)A_2(z_\nu)u(z_\nu) \\ + m_\nu^2 A_3(z_\nu)u^2(z_\nu) \end{array}\right] u(z_\nu)}{\begin{array}{l} \frac{1}{4!}(3m_\nu^3+15m_\nu^2+5m_\nu+1) \\ - \frac{m_\nu}{12}(7m_\nu^2+24m_\nu+5)A_2(z_\nu)u(z_\nu) \\ + \frac{m_\nu^2}{2}[(m_\nu+3)A_3(z_\nu)+(m_\nu+1)A_2^2(z_\nu)]u^2(z_\nu) \\ - m_\nu^3 A_2(z_\nu)A_3(z_\nu)u^3(z_\nu) \\ + \frac{m_\nu^3}{3}A_2^3(z_\nu)u^3(z_\nu) \\ - \frac{m_\nu^2}{3}T_3(z_\nu)u^3(z_\nu) \end{array}} \quad . \tag{B.17}$$

This final equation is the rational form of the variable-order IF for multiple zeros given in Equation (5.53) on page 61.

$$\hat{z}_\nu = z_\nu - \frac{m_\nu u(z_\nu)}{1 - \sum\limits_{i\neq\nu} \frac{m_i}{z_\nu - [\hat{z}_i = z_i - m_i u(z_i)]}} \quad . \tag{B.18}$$

Appendix C

Program Listings

This appendix brings together all the details concerning the implementation of our algorithms described in Chapter 5 starting on page 42 in the form of computer programs and scripts in various languages.

The line numbers given here are not part of the files listed. They are included for reference purposes only.

C.1 Introduction

C.1.1 Our Equipment

In the early days of our work on the zeros of polynomials we used the ICT[1] Atlas 1 computer [Sti72], that was housed in Gordon Square, London, England and purchased jointly by a consortium of the University of London and British Petroleum and installed in 1964.

For a look back to these early days, see [Cro12].

We later migrated to a CDC 6600 computer housed in the University of London Computer Centre (ULCC) building in Guilford Street, London. Although state-of-the-art at the time, our computations in Fortran's `DOUBLE PRECISION` arithmetic took so long that the iteration stage (all we had at the time) needed checkpoint/restart facilities over a number of days before convergence was achieved.

Now fast forward.

[1] Later to become ICL

Table C.1 on page 126 summarises the hardware used for our current computations.

Component	Description
Cache	6 MB
Cores	4
CPU	Intel Core i5-3470
Disk	500 GB
Memory	8 GB
Model	Viglen Genie (Stone PC-1103)

Table C.1: Our Hardware

Table C.2 on page 126 summarises the current software used for our computations.

Component	Description
Kernel	Linux 3.8.4-102
Languages	GCC 4.7.2 and Matlab 8.0.0.783
Library	GMP 5.0.2
OS	Fedora 17 x86_64

Table C.2: Our Software

For details of the GMP library, see §C.1.3 on page 127, and for details of Matlab, see §C.1.4 starting on page 127.

This arrangement allows us to run both the search stage and the iterative stage in real time taking at most a few minutes to compute the required zeros to our required accuracy.

That is progress.

C.1.2 Program Options

We have been writing computer programs for many decades now and still find the traditional UNIX-style command line interface the most flexible programming environment available.

Therefore, it is important that the programs we invoke on the command line have a consistent look and feel. This is especially important for the programs that implement the search stage (see §5.5.1 on page 62) and the iterative stage (see §5.5.2 on page 63). The command line options provide this consistency.

Table C.3 on page 127 shows the options relevant to our workhorse script `solve.sh` (see §C.7.3 on page 209).

Option	Description
-c *n*	Consolidation multiplier (default 3). See §5.2.2 on page 47.
-d *n*	Debug search stage. See Table C.4 on page 127.
-D *n*	Debug iterative stage. See Table C.4 on page 127.
-i	Iterate only, use search stage results as inputs.
-l *polys*	List of polynomials (default `$failure`).
-m *n*	Number of outer iterations (default 8).
-n *n*	Number of inner iterations per outer iteration (default 4).
-o	Turn off options (text) within polynomials.
-p *n*	Precision during search stage (default 128 bits).
-P *n*	Precision during iterative stage (default 128 bits).
-s *n*	Sides during search stage (default 2 for 4 squares).
-S	Use `$success` list of polynomials.
-z *n*	Zeros available during search stage (default `5*DEGREE(myp)`).

Table C.3: Script Options

Table C.4 on page 127 shows the debug options available. When set, these options trace certain places and output additional values. Options can be combined by adding their values together.

Option	Description
1	Polynomial evaluation. See `mylib.c` in §C.3.6 on page 144.
2	Newton evaluation. See `mylib.c` in §C.3.6 on page 144.
4	Complex division. See `complex.c` in §C.3.1 on page 134.

Table C.4: Debug Options

C.1.3 GNU Multiple Precision Arithmetic Library (GMP)

The GNU Multiple Precision Arithmetic Library [Gra11], often referred to as GMP in the literature, is a library of functions written in C, callable from other C, and C-compatible, programs. It enables programmers to store and manipulate numerical values within a specified precision, chosen at run-time.

C.1.4 Matlab

Matlab is a computing environment that attempts to provide the ideal environment for developing solutions to problems in multiple disciplines[Mat13]. We quote from their web site.

> "Matlab is a high-level language and interactive environment for numerical computation, visualization, and programming. Using Matlab, you can analyze data, develop algorithms, and create models

and applications. The language, tools, and built-in math functions enable you to explore multiple approaches and reach a solution faster than with spreadsheets or traditional programming languages, such as C/C++ or Java.

"You can use Matlab for a range of applications, including signal processing and communications, image and video processing, control systems, test and measurement, computational finance, and computational biology. More than a million engineers and scientists in industry and academia use Matlab, the language of technical computing."

Our Matlab programs are listed in §C.6 starting on page 190.
REMARK 14. *C code listings not provided. Winkler point 12.*

C.2 C Header Files

This section describes all the C header files used by the programs for implementing both stage 1 and stage 2 of our algorithm.

C.2.1 complex.h

This header file defines our `Complex` type which is a structure containing two `Real` numbers. They are accessed by the macros `REAL()` and `IMAG()`. It also contains the prototypes of the complex arithmetic functions.

```
1   #include "real.h"
2
3   #ifndef _COMPLEX_
4
5   #define _COMPLEX_
6   #define REAL(c) (c->real)
7   #define IMAG(c) (c->imag)
8
9   typedef struct complex {
10    Real real;
11    Real imag;
12  } Complex[1];
13
14  extern void cabs(Real, const Complex);
15  extern void cadd(Complex, const Complex, const Complex);
16  extern void caddi(Complex, const Complex, const int);
17  extern int  ccmp(const Complex, const Complex);
```

```
18  extern void cconj(Complex, const Complex);
19  extern void ccopy(Complex, const Complex);
20  extern void cdiv(Complex, const Complex, const Complex);
21  extern void cdivi(Complex, const Complex, const int);
22  extern void cdivr(Complex, const Complex, const Real);
23  extern void cdump(const Complex);
24  extern void cerror(Real, const Complex, const Complex);
25  extern void cfree(Complex);
26  extern void ciadd(Complex, const int, const Complex);
27  extern void cimul(Complex, const int, const Complex);
28  extern void cinput(Complex);
29  extern void ciota(Complex);
30  extern void cisub(Complex, const int, const Complex);
31  extern void cmul(Complex, const Complex, const Complex);
32  extern void cneg(Complex, const Complex);
33  extern void cnew(Complex, const double, const double);
34  extern void coutput(const Complex);
35  extern void crmul(Complex, const Real, const Complex);
36  extern void cset(Complex, const double, const double);
37  extern void csub(Complex, const Complex, const Complex);
38  extern void cswap(Complex, Complex);
39
40  #endif
```

C.2.2 farmer3.h et al.

The header files for the IFs we use to demonstrate our approach are identical except for the name of the IF, e.g. `ehrlich3.h, farmer3.h`, etc. so a single sample is given here to show the structure.

This header file contains the prototype of the functions required to run one specific IF, namely our third-order IF.

```
1   #include "complex.h"
2   #include "zero.h"
3
4   #ifndef _FARMER3_
5
6   #define _FARMER3_
7
8   void farmer3(Zero, const Zero, const char);
9
10  #endif
```

C.2.3 mylib.h

This header file contains a couple of default constants and a number of global variables in addition to the prototypes of miscellaneous functions used by our programs, but mainly those used during the iterative stage.

```
1   #include <math.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include "poly.h"
5   #include "zero.h"
6
7   #ifndef _MYLIB_
8
9   #define _MYLIB_
10  #define MAX_BUFFER 256
11  #define MAX_ITERS  16
12
13  extern int  count;
14  extern int  debug;
15  extern int *done;
16  extern int  precision;
17
18  void A2(Complex, const Complex);
19  void A3(Complex, const Complex);
20  void error(char*, ...);
21  void fatal(char*, ...);
22  int  getinteger(void);
23  int  iinput(void);
24  int  iterate(Zero, const Zero,
25               void (*)(Zero, const Zero, const char), const char);
26  void newton2(Complex, const Complex, const int);
27  int  run(int, char *[],
28           void (*)(Zero, const Zero, const char), const char *);
29  void sighandler(int);
30  void T1(Complex, const Zero, const int);
31  void T1n(Complex, Zero, const int);
32  void T2(Complex, const Zero, const int);
33  void T3(Complex, const Zero, const int);
34  void u(Complex, const Complex);
35  void u1(Complex, const Complex);
36
37  #endif
```

C.2.4 poly.h

This header file defines our `Poly` type which is a vector of `Complex` numbers indexed from zero to the macro `DEGREE()`. Individual coefficients are accessed by the `COEFF()` macro and the size originally allocated to the polynomial by the `PSIZE()` macro. It also contains the prototypes of the polynomial arithmetic functions.

```
1   #include "complex.h"
2
3   #ifndef _POLY_
4
5   #define _POLY_
6   #define COEFF(p, i) (p->coeff[i])
7   #define DEGREE(p)   (p->degree)
8   #define PSIZE(p)    (p->size)
9
10  typedef struct poly {
11    Complex *coeff;
12    int      degree; /* current degree */
13    int      size;   /* declared size of array */
14  } Poly[1];
15
16  extern Poly myp;   /* Global polynomial */
17  extern Poly myp1;  /* First derivative */
18  extern Poly myp2;  /* Second derivative */
19  extern Poly myp3;  /* Third derivative */
20
21  extern void padd(Poly, const Poly, const Poly);
22  extern void pcopy(Poly, const Poly);
23  extern int  pdegree(const Poly);
24  extern void pdiff(Poly, const Poly);
25  extern void pdivi(Poly, const Poly, const int);
26  extern void pdivr(Poly, const Poly, const Real);
27  extern void pdump(const Poly);
28  extern void peval(Complex, const Poly, const Complex);
29  extern void pfree(Poly);
30  extern void pimul(Poly, const int, const Poly);
31  extern void pinput(Poly);
32  extern void pmul(Poly, const Poly, const Poly);
33  extern void pnew(Poly, const int);
34  extern void poutput(const Poly);
35  extern void pscale(Poly, const Poly, const double);
36  extern void psetup(Poly, const Poly);
37  extern void pshift(Poly, const Poly, const Complex);
38  extern void psmooth(Poly, const Poly);
39  extern void psub(Poly, const Poly, const Poly);
```

```
40
41  #endif
```

C.2.5 real.h

This header file defines our `Real` type which is mapped onto a multiple precision floating-point number from the GNU Multiple Precision Arithmetic Library (GMP) [Gra11]. It also contains the prototypes of the real arithmetic functions.

```
1   #include <float.h>
2   #include <gmp.h>
3
4   #ifndef _REAL_
5
6   #define _REAL_
7   /*
8    * IEEE 754 precisions
9    * Single: 32-bit word, 24-bit mantissa (10 decimal digits)
10   * Double: 64-bit word, 53-bit mantissa (17 decimal digits)
11   */
12
13  #define REAL_EPSILON   1e-24
14  #define REAL_FORMAT    "% 25.24Fe"
15  #define REAL_IOTA      1e-96
16  #define REAL_PRECISION 128
17
18  typedef mpf_t Real;
19
20  extern void   rabs(Real, const Real);
21  extern void   radd(Real, const Real, const Real);
22  extern int    rcmp(const Real, const Real);
23  extern int    rcmpd(const Real, const double);
24  extern void   rcopy(Real, const Real);
25  extern void   rdiv(Real, const Real, const Real);
26  extern void   rdump(const Real);
27  extern void   rfree(Real);
28  extern double rget(const Real);
29  extern int    rinit(const int);
30  extern void   rinput(Real);
31  extern void   rmul(Real, const Real, const Real);
32  extern void   rneg(Real, const Real);
33  extern void   rnew(Real, const double);
34  extern void   riota(Real);
35  extern void   routput(const Real);
36  extern void   rset(Real, const double);
37  extern void   rsqrt(Real, const Real);
38  extern void   rsub(Real, const Real, const Real);
```

```
39  extern void   rswap(Real, Real);
40
41  #endif
```

C.2.6 search.h

This header file defines various default values used during the search stage. It also contains the prototypes of the functions used during the search stage.

```
1   #include "poly.h"
2   #include "zero.h"
3
4   #define COVER 3
5   #define MAX1  8
6   #define MAX2  4
7   #define SIDES 2
8
9   extern double dekker(const Poly);
10  extern int    marden(const Poly, const Complex, const double);
11  extern double pave(Zero, const Poly, const Complex,
12                     const double,  const int);
13  extern void   reduce(Poly, const Poly);
14  extern int    search(Zero, const Poly, const int, const int,
15                       const int, const double);
16  extern void   test(Zero, const Poly, const Complex, const double);
```

C.2.7 zero.h

This header file defines our `Zero` type which is a vector of structures containing approximations to the zeros of a polynomial together with their multiplicities, indexed from zero to `ZEROS()`. Individual approximations are accessed by the `ZERO()` macro and their multiplicity by the `MULT()` macro. The size of the vector is accessed by the `ZSIZE()` macro. It also contains the prototypes of the zero arithmetic functions.

```
1   #include "complex.h"
2
3   #ifndef _ZERO_
4
5   #define _ZERO_
6   #define MULT(z, i) (z->mult[i])
7   #define ZSIZE(z)   (z->size)
8   #define ZERO(z, i) (z->zero[i])
9   #define ZEROS(z)   (z->count)
```

```
10
11  typedef struct zero {
12    int      count; /* current size */
13    int     *mult;  /* current multiplicities */
14    int      size;  /* declared size of the array */
15    Complex *zero;  /* current zeros */
16  } Zero[1];
17
18  extern void zcopy(Zero, const Zero);
19  extern void zcover(Zero, const Zero, const double, const double);
20  extern int  zdegree(const Zero);
21  extern void zdump(const Zero);
22  extern void zfree(Zero);
23  extern void zinput(Zero);
24  extern void znew(Zero, const int);
25  extern void ziota(Zero);
26  extern void zoutput(const Zero);
27  extern int  zpartition(Zero, const int, const int, const int);
28  extern void zsort(Zero, const int, const int);
29  extern void zswap(Zero, const int lhs, const int rhs);
30
31  #endif
```

C.3 C Program Files

This section describes all the C program files used by the programs for implementing both stage 1 and stage 2 of our algorithm.

C.3.1 complex.c

This program file contains the definitions of the functions for manipulating `Complex` values. They are based on the functions for manipulating `Real` values described in §C.3.9, starting on page 161.

```
1   #include "complex.h"
2   #include "mylib.h"
3
4   /*
5    * Compute the absolute value of a Complex number
6    */
7   void cabs(Real out, const Complex in) {
8     Real rtmp1;
9     Real rtmp2;
10    rnew(rtmp1, 0);
```

```
11    rnew(rtmp2, 0);
12    rmul(rtmp1, REAL(in), REAL(in));
13    rmul(rtmp2, IMAG(in), IMAG(in));
14    radd(out, rtmp1, rtmp2);
15    rsqrt(out, out);
16    rfree(rtmp1);
17    rfree(rtmp2);
18  }
19
20  /*
21   * Add two Complex numbers
22   */
23  void cadd(Complex out, const Complex lhs, const Complex rhs) {
24    radd(REAL(out), REAL(lhs), REAL(rhs));
25    radd(IMAG(out), IMAG(lhs), IMAG(rhs));
26  }
27
28  /*
29   * Add a Complex number and an integer
30   */
31  void caddi(Complex out, const Complex lhs, const int rhs) {
32    Real rtmp;
33    rnew(rtmp, rhs);
34    radd(REAL(out), REAL(lhs), rtmp);
35    rcopy(IMAG(out), IMAG(lhs));
36    rfree(rtmp);
37  }
38
39  /*
40   * Compare two Complex numbers.  Arbitrary ordering for
41   * comparison purposes
42   */
43  int ccmp(const Complex lhs, const Complex rhs) {
44    int tmp = rcmp(REAL(lhs), REAL(rhs));
45    if (tmp != 0) {
46      return tmp;
47    }
48    return rcmp(IMAG(lhs), IMAG(rhs));
49  }
50
51  /*
52   * Compute the complex conjugate
53   */
54  void cconj(Complex out, const Complex in) {
55    rcopy(REAL(out), REAL(in));
56    rneg(IMAG(out), IMAG(in));
57  }
58
59  /*
60   * Copy a Complex number
```

```
61    */
62   void ccopy(Complex out, const Complex in) {
63     rcopy(REAL(out), REAL(in));
64     rcopy(IMAG(out), IMAG(in));
65   }
66
67   /*
68    * Complex division using brute force
69    */
70   void cdiv(Complex out, const Complex numer, const Complex denom) {
71     Complex ctmp;
72     Real    rtmp;
73     cnew(ctmp, 0, 0);
74     rnew(rtmp, 0);
75     cconj(ctmp, denom);
76     cmul(ctmp, numer, ctmp);
77     cabs(rtmp, denom);
78     cdivr(out, ctmp, rtmp);
79     cdivr(out, out, rtmp);
80     if (debug & 4) {
81       printf("Debug4: Cdiv\nNumer\n");
82       cdump(numer);
83       printf("\nDenom\n");
84       cdump(denom);
85       printf("\nOutput\n");
86       cdump(out);
87       printf("\n");
88     }
89     cfree(ctmp);
90     rfree(rtmp);
91   }
92
93   /*
94    * Divide a Complex number by an integer
95    */
96   void cdivi(Complex out, const Complex numer, const int denom) {
97     Real rtmp;
98     rnew(rtmp, denom);
99     cdivr(out, numer, rtmp);
100    rfree(rtmp);
101  }
102
103  /*
104   * Divide a Complex number by a Real
105   */
106  void cdivr(Complex out, const Complex numer, const Real denom) {
107    rdiv(REAL(out), REAL(numer), denom);
108    rdiv(IMAG(out), IMAG(numer), denom);
109  }
110
```

```
111  /*
112   *  Dump a Complex number to the highest possible accuracy
113   */
114  void cdump(const Complex in) {
115    rdump(REAL(in));
116    printf(" ");
117    rdump(IMAG(in));
118  }
119
120  /*
121   * Compute relative error of two Complex numbers
122   */
123  void cerror(Real out, const Complex lhs, const Complex rhs) {
124    Complex ctmp;
125    Real    denom;
126    Real    numer;
127    cnew(ctmp, 0, 0);
128    rnew(denom, 0);
129    rnew(numer, 0);
130    csub(ctmp, lhs, rhs);
131    cabs(numer, ctmp);
132    cabs(denom, lhs);
133    rdiv(out, numer, denom);
134    cfree(ctmp);
135    rfree(denom);
136    rfree(numer);
137  }
138
139  /*
140   * Free the storage for a Complex number
141   */
142  void cfree(Complex in) {
143    rfree(REAL(in));
144    rfree(IMAG(in));
145  }
146
147  /*
148   * Add an integer and a Complex number
149   */
150  void ciadd(Complex out, const int lhs, const Complex rhs) {
151    Real tmp;
152    rnew(tmp, lhs);
153    radd(REAL(out), tmp, REAL(rhs));
154    rcopy(IMAG(out), IMAG(rhs));
155    rfree(tmp);
156  }
157
158  /*
159   * Multiple an integer by a Complex number
160   */
```

```
161  void cimul(Complex out, const int lhs, const Complex rhs) {
162    Real tmp;
163    rnew(tmp, lhs);
164    crmul(out, tmp, rhs);
165    rfree(tmp);
166  }
167
168  /*
169   * Input a Complex number
170   */
171  void cinput(Complex out) {
172    rinput(REAL(out));
173    rinput(IMAG(out));
174  }
175
176  /*
177   * Set very small Complex numbers to zero
178   */
179  void ciota(Complex c) {
180    riota(REAL(c));
181    riota(IMAG(c));
182  }
183
184  /*
185   * Subtract a Complex number from an integer
186   */
187  void cisub(Complex out, const int lhs, const Complex rhs) {
188    Real tmp;
189    rnew(tmp, lhs);
190    rsub(REAL(out), tmp, REAL(rhs));
191    rcopy(IMAG(out), IMAG(rhs));
192    rfree(tmp);
193  }
194
195  /*
196   * Multiple two Complex numbers
197   */
198  void cmul(Complex out, const Complex lhs, const Complex rhs) {
199    Complex ctmp;
200    Real    rtmp1;
201    Real    rtmp2;
202    cnew(ctmp, 0, 0);
203    rnew(rtmp1, 0);
204    rnew(rtmp2, 0);
205    rmul(rtmp1, REAL(lhs), REAL(rhs));
206    rmul(rtmp2, IMAG(lhs), IMAG(rhs));
207    rsub(REAL(ctmp), rtmp1, rtmp2);
208    rmul(rtmp1, REAL(lhs), IMAG(rhs));
209    rmul(rtmp2, IMAG(lhs), REAL(rhs));
210    radd(IMAG(ctmp), rtmp1, rtmp2);
```

```
211     ccopy(out, ctmp);
212     cfree(ctmp);
213     rfree(rtmp1);
214     rfree(rtmp2);
215   }
216
217   /*
218    * Multiple a Real by a Complex number
219    */
220   void crmul(Complex out, const Real lhs, const Complex rhs) {
221     rmul(REAL(out), lhs, REAL(rhs));
222     rmul(IMAG(out), lhs, IMAG(rhs));
223   }
224
225   /*
226    * Negate a Complex number
227    */
228   void cneg(Complex out, const Complex in) {
229     rneg(REAL(out), REAL(in));
230     rneg(IMAG(out), IMAG(in));
231   }
232
233   /*
234    * Create a new Complex number from two double precision
235    * numbers
236    */
237   void cnew(Complex out, const double re, const double im) {
238     rnew(REAL(out), re);
239     rnew(IMAG(out), im);
240   }
241
242   /*
243    * Output a Complex number
244    */
245   void coutput(const Complex in) {
246     routput(REAL(in));
247     printf(" ");
248     routput(IMAG(in));
249   }
250
251   /*
252    * Create a Complex number from two double precision numbers
253    */
254   void cset(Complex out, const double re, const double im) {
255     rset(REAL(out), re);
256     rset(IMAG(out), im);
257   }
258
259   /*
260    * Subtract two Complex numbers
```

```
261    */
262   void csub(Complex out, const Complex lhs, const Complex rhs) {
263     rsub(REAL(out), REAL(lhs), REAL(rhs));
264     rsub(IMAG(out), IMAG(lhs), IMAG(rhs));
265   }
266
267   /*
268    * Swap two Complex numbers
269    */
270   void cswap(Complex lhs, Complex rhs) {
271     rswap(REAL(lhs), REAL(rhs));
272     rswap(IMAG(lhs), IMAG(rhs));
273   }
```

C.3.2 ehrlich3.c

This program file contains the function for evaluating Louis Ehrlich's third-order IF defined in Equation (B.15) on page 123.

```
1    #include <stdio.h>
2    #include "mylib.h"
3    #include "ehrlich3.h"
4    #include "zero.h"
5
6    int main(int argc, char *argv[]) {
7      return run(argc, argv, &ehrlich3, "Ehrlich's third-order");
8    }
9
10   /*
11    * Perform one iteration of Ehrlich's third-order method
12    */
13   void ehrlich3(Zero out, const Zero in, const char mode) {
14     Complex ctmp;
15     Complex denom;
16     int     nu;
17     Complex numer;
18     Complex utmp;
19     Zero    z;
20     cnew(ctmp, 0, 0);
21     cnew(denom, 0, 0);
22     cnew(numer, 0, 0);
23     cnew(utmp, 0, 0);
24     znew(z, ZEROS(in));
25     zcopy(z, in);
26     for (nu = 0; nu < ZEROS(z); nu++) {
27       if (done[nu]) {
28         continue;
```

```
29        }
30        u(utmp, ZERO(z, nu));
31        cimul(numer, MULT(z, nu), utmp);
32        T1(ctmp, z, nu);
33        cmul(ctmp, ctmp, utmp);
34        cisub(denom, 1, ctmp);
35        cdiv(ctmp, numer, denom);
36        csub(ZERO(out, nu), ZERO(z, nu), ctmp);
37        if (mode == 's') {
38          ccopy(ZERO(z, nu), ZERO(out, nu));
39        }
40      }
41      cfree(ctmp);
42      cfree(denom);
43      cfree(numer);
44      cfree(utmp);
45      zfree(z);
46    }
```

C.3.3 farmer3.c

This program file contains the function for evaluating Mick Farmer and George Loizou's third-order IF defined in Equation (5.49) on page 60.

```
1   #include <stdio.h>
2   #include "mylib.h"
3   #include "farmer3.h"
4   #include "zero.h"
5
6   int main(int argc, char *argv[]) {
7     return run(argc, argv, &farmer3, "Farmer's third-order (multiple zeros)");
8   }
9
10  /*
11   * Perform one iteration of Farmer's third-order multiple
12   * zero method (p. 139).  Coded for clarity rather than
13   * efficiency.
14   */
15  void farmer3(Zero out, const Zero in, const char mode) {
16    Complex cexpr;
17    Complex cterm;
18    int     nu;
19    Complex utemp;
20    Zero    z;
21    cnew(cexpr, 0, 0);
22    cnew(cterm, 0, 0);
23    cnew(utemp, 0, 0);
```

```
24      znew(z, ZEROS(in));
25      zcopy(z, in);
26      for (nu = 0; nu < ZEROS(z); nu++) {
27        if (done[nu]) {
28          continue;
29        }
30        u(utemp, ZERO(z, nu));
31        /* Step 1 */
32        cimul(cexpr, MULT(z, nu), utemp);
33        if (debug & 8) {
34          printf("Correction 1\n");
35          coutput(cexpr);
36          printf("\n");
37        }
38        /* Step 2 */
39        T1(cterm, z, nu);
40        cimul(cterm, MULT(z, nu), cterm);
41        cmul(cterm, cterm, utemp);
42        cmul(cterm, cterm, utemp);
43        cadd(cexpr, cexpr, cterm);
44        if (debug & 8) {
45          printf("Correction 2\n");
46          coutput(cexpr);
47          printf("\n");
48        }
49        /* Step 3 */
50        csub(ZERO(out, nu), ZERO(z, nu), cexpr);
51        if (mode == 's') {
52          ccopy(ZERO(z, nu), ZERO(out, nu));
53        }
54      }
55      cfree(cexpr);
56      cfree(cterm);
57      cfree(utemp);
58      zfree(z);
59    }
```

C.3.4 farmerv.c

This program file contains the function for evaluating Mick Farmer and George Loizou's variable-order IF defined in Equation (5.53) on page 61.

```
1   #include <stdio.h>
2   #include "farmerv.h"
3   #include "mylib.h"
4   #include "zero.h"
5
```

```
6   int main(int argc, char *argv[]) {
7     return run(argc, argv, &farmerv, "Farmer's variable-order");
8   }
9
10  /*
11   * Perform one iteration of Farmer's variable-order IF
12   * (based on Farmer's third-order)
13   */
14  void farmerv(Zero out, const Zero in, const char mode) {
15    Complex csum;
16    Complex ctmp;
17    int     nu;
18    Complex mutmp;
19    cnew(csum, 1, 0);
20    cnew(ctmp, 0, 0);
21    cnew(mutmp, 0, 0);
22    zcopy(out, in);
23    for (nu = 0; nu < ZEROS(out); nu++) {
24      if (done[nu]) {
25        continue;
26      }
27      u(mutmp, ZERO(out, nu));
28      cimul(mutmp, MULT(out, nu), mutmp);
29      T1n(ctmp, out, nu);
30      cmul(ctmp, ctmp, mutmp);
31      csub(csum, csum, ctmp);
32      cmul(csum, mutmp, csum);
33      csub(ZERO(out, nu), ZERO(out, nu), csum);
34    }
35    cfree(csum);
36    cfree(ctmp);
37    cfree(mutmp);
38  }
```

C.3.5 hansen3.c

This program file contains the function for evaluating Vagn Hansen's third-order IF defined in Equation (B.10) on page 122.

```
1   #include <stdio.h>
2   #include "hansen3.h"
3   #include "mylib.h"
4   #include "zero.h"
5
6   int main(int argc, char *argv[]) {
7     return run(argc, argv, &hansen3, "Hansen3's third-order");
8   }
```

```
9
10  /*
11   * Perform one iteration of Hansen's third-order method
12   */
13  void hansen3(Zero out, const Zero in, const char mode) {
14    Complex ctmp;
15    Complex denom;
16    int     nu;
17    Complex numer;
18    Complex utmp;
19    Zero    z;
20    cnew(ctmp, 0, 0);
21    cnew(denom, 0, 0);
22    cnew(numer, 0, 0);
23    cnew(utmp, 0, 0);
24    znew(z, ZEROS(in));
25    zcopy(z, in);
26    for (nu = 0; nu < ZEROS(z); nu++) {
27      if (done[nu]) {
28        continue;
29      }
30      u(utmp, ZERO(z, nu));
31      cimul(numer, MULT(z, nu), utmp);
32      cset(denom, 0.5*(1+MULT(z, nu)), 0);
33      A2(ctmp, ZERO(z, nu));
34      cmul(ctmp, numer, ctmp);
35      csub(denom, denom, ctmp);
36      cdiv(ctmp, numer, denom);
37      csub(ZERO(out, nu), ZERO(z, nu), ctmp);
38    }
39    cfree(ctmp);
40    cfree(denom);
41    cfree(numer);
42    cfree(utmp);
43    zfree(z);
44  }
```

C.3.6 mylib.c

This program file contains the definitions of various functions that do not fit in with the more focused program files. These vary from functions used in both stages of our algorithm, e.g. reporting errors or evaluating $u(z)$ and its derivatives, or those evaluating useful functions, e.g. $A_i(z)$ and $T_i(z)$ which are used in stage two.

```
1   /*
```

```
 * My library of common functions used by other modules
 */
#include <getopt.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "mylib.h"
#include "poly.h"
#include "zero.h"

int  count     = -1; /* Count of convergent approximations.
                      * A negative value indicates that we
                      * are in the search stage, where we do
                      * do not call ciota() when evaluating
                      * u(z), see below, because the
                      * function lagouanelle() requires
                      * accurate values, not truncated
                      * towards zero values which suit our
                      * IFs. */
int  debug     = 0;  /* Debugging flags */
int *done;           /* Approximation has converged */
int  precision = REAL_PRECISION;

/*
 * Compute the value of A2(z)
 */
void A2(Complex out, const Complex in) {
  Complex denom;
  Complex numer;
  cnew(denom, 0, 0);
  cnew(numer, 0, 0);
  peval(numer, myp2, in);
  peval(denom, myp1, in);
  cimul(denom, 2, denom);
  cdiv(out, numer, denom);
  cfree(denom);
  cfree(numer);
}

/*
 * Compute the value of A3(z)
 */
void A3(Complex out, const Complex in) {
  Complex denom;
  Complex numer;
  cnew(denom, 0, 0);
  cnew(numer, 0, 0);
  peval(numer, myp3, in);
```

```
52      peval(denom, myp1, in);
53      cimul(denom, 6, denom);
54      cdiv(out, numer, denom);
55      cfree(denom);
56      cfree(numer);
57    }
58
59    /*
60     * Report an error on standard error
61     */
62    void error(char *fmt, ...) {
63      va_list args;
64      va_start(args, fmt);
65      vfprintf(stderr, fmt, args);
66      fprintf(stderr, "\n");
67      va_end(args);
68    }
69
70    /*
71     * Report a fatal error on standard error, then exit
72     */
73    void fatal(char *fmt, ...) {
74      va_list args;
75      va_start(args, fmt);
76      vfprintf(stderr, fmt, args);
77      fprintf(stderr, "\n");
78      va_end(args);
79      exit(EXIT_FAILURE);
80    }
81
82    /*
83     * Input an integer, stripping blank lines or leading
84     * comments
85     */
86    int getinteger(void) {
87      char buff[MAX_BUFFER];
88      int  i;
89      int  n;
90      /* Strip leading comments and blank lines */
91      do {
92        fgets(buff, sizeof(buff), stdin);
93        i = strspn(buff, " ");
94      } while (buff[i] == '#' || buff[i] == '\n');
95      if (sscanf(buff+i, "%d", &n) != 1) {
96        fatal("Getinteger error (%s)", buff);
97      }
98      return n;
99    }
100
101   /*
```

```
102    * Input an integer
103    */
104   int iinput(void) {
105     int n;
106     if (scanf("%d", &n) != 1) {
107       fatal("Iinput read error");
108     }
109     return n;
110   }
111
112   /*
113    * Perform a number of iterations of an IF
114    */
115   int iterate(Zero out, const Zero in,
116               void (*ftn)(Zero, const Zero, const char),
117               const char mode) {
118     int     iter;
119     int     nu;
120     Real    rtmp;
121     clock_t tstart = clock();
122     Zero    z;
123     count = 0;
124     done = calloc(ZEROS(in), sizeof(int));
125     rnew(rtmp, 0);
126     znew(z, ZEROS(in));
127     zcopy(z, in);
128     for (iter = 1; count < ZEROS(z) && iter <= MAX_ITERS; iter++) {
129       printf("Iteration %d\n", iter);
130       (*ftn)(out, z, mode);
131       zoutput(out);
132       for (nu = 0; nu < ZEROS(z); nu++) {
133         if (done[nu]) {
134           continue;
135         }
136         cerror(rtmp, ZERO(out, nu), ZERO(z, nu));
137         printf("Err[%2d] ", nu+1);
138         routput(rtmp);
139         if (rcmpd(rtmp, REAL_EPSILON) < 0) {
140           count++;
141           done[nu] = 1;
142           printf("  <==");
143         }
144         printf("\n");
145       }
146       printf("Convergence %d %d\n", iter, count);
147       zcopy(z, out);
148     }
149     free(done);
150     rfree(rtmp);
151     zfree(z);
```

```
152     printf("ITERATE TIME %f\n",
153            (double)(clock()-tstart)/CLOCKS_PER_SEC);
154     if (count == ZEROS(z)) {
155       FILE *fd;
156       if ((fd = fopen("/tmp/iters", "w")) == NULL) {
157         fatal("Iterate can't open temporary file");
158       }
159       fprintf(fd, "%d\n", iter-1);
160       fclose(fd);
161        rinit(REAL_PRECISION);
162       znew(z, ZEROS(z));
163       zcopy(z, out);
164       ziota(z);
165       zsort(z, 0, ZEROS(z)-1);
166       zoutput(z);
167       return(EXIT_SUCCESS);
168     }
169     return(EXIT_FAILURE);
170   }
171
172   /*
173    * Compute a Newton second-order approximation
174    */
175   void newton2(Complex out, const Complex in, const int mult) {
176     Complex ctmp;
177     cnew(ctmp, 0, 0);
178     u(ctmp, in);
179     cimul(ctmp, mult, ctmp);
180     csub(out, in, ctmp);
181     if (debug & 2) {
182       printf("Debug2: Newton2\nInput\n");
183       cdump(in);
184       printf("\nOutput\n");
185       cdump(out);
186       printf("\n");
187     }
188     cfree(ctmp);
189   }
190
191   /*
192    * Run an IF
193    */
194   int run(int argc, char *argv[],
195            void (*ftn)(Zero, const Zero, const char),
196            const char *name) {
197     int  c;
198     Zero in;
199     char mode = 'p';
200     Zero out;
201     precision = 4*REAL_PRECISION;
```

```
202     while ((c = getopt(argc, argv, "d:p:s")) != -1) {
203       switch (c) {
204       case 'd':
205         debug = atoi(optarg);
206         break;
207       case 'p':
208         precision = atoi(optarg);
209         break;
210       case 's':
211         mode = 's';
212         break;
213       default:
214         fatal("Unknown option %s", argv[optind-1]);
215       }
216     }
217     rinit(precision);
218     printf("%s (%c)\n", name, mode);
219     pinput(myp);
220     printf("Polynomial\n");
221     poutput(myp);
222     psetup(myp, myp);
223     printf("Monic polynomial\n");
224     poutput(myp);
225     pdiff(myp1, myp);
226     printf("First derivative\n");
227     poutput(myp1);
228     pdiff(myp2, myp1);
229     printf("Second derivative\n");
230     poutput(myp2);
231     pdiff(myp3, myp2);
232     printf("Third derivative\n");
233     poutput(myp3);
234     zinput(in);
235     printf("Initial approximations\n");
236     zoutput(in);
237     if (DEGREE(myp) != zdegree(in)) {
238       fatal("Iterate run degree mismatch, poly %d, zeros %d",
239             DEGREE(myp), zdegree(in));
240     }
241     znew(out, ZEROS(in));
242     for (c = 0; c < ZEROS(out); c++) {
243       MULT(out, c) = MULT(in, c);
244     }
245     return iterate(out, in, ftn, mode);
246   }
247
248   /*
249    * Handle a signal, presumably from a failed
250    * assertion
251    */
```

```
252  void sighandler(int sibnum) {
253    fprintf(stderr, "ABORT!\n");
254    exit(EXIT_FAILURE);
255  }
256
257  /*
258   * Compute the value of T1(z_nu)
259   */
260  void T1(Complex out, const Zero in, const int nu) {
261    Complex denom;
262    int     i;
263    Complex numer;
264    Complex sum;
265    cnew(denom, 0, 0);
266    cnew(numer, 0, 0);
267    cnew(sum, 0, 0);
268    for (i = 0; i < ZEROS(in); i++) {
269      if (i == nu) {
270        continue;
271      }
272      cset(numer, MULT(in, i), 0);
273      csub(denom, ZERO(in, nu), ZERO(in, i));
274      cdiv(numer, numer, denom);
275      cadd(sum, sum, numer);
276    }
277    ccopy(out, sum);
278    cfree(denom);
279    cfree(numer);
280    cfree(sum);
281  }
282
283  /*
284   * Compute the value of T1n(z_nu)
285   */
286  void T1n(Complex out, Zero in, const int nu) {
287    Complex ctmp;
288    Complex denom;
289    int     i;
290    Complex numer;
291    Real    rtmp;
292    Complex sum;
293    cnew(ctmp, 0, 0);
294    cnew(denom, 0, 0);
295    cnew(numer, 0, 0);
296    rnew(rtmp, 0);
297    cnew(sum, 0, 0);
298    for (i = 0; i < ZEROS(in); i++) {
299      if (i == nu) {
300        continue;
301      }
```

```
302       cset(numer, MULT(in, i), 0);
303       if (!done[i]) {
304         newton2(ctmp, ZERO(in, i), MULT(in, i));
305         cerror(rtmp, ctmp, ZERO(in, i));
306         if (rcmpd(rtmp, REAL_EPSILON) < 0) {
307           count++;
308           done[i] = 1;
309           printf("Err[%2d] ", i+1);
310           routput(rtmp);
311           printf(" <==\n");
312         }
313         ccopy(ZERO(in, i), ctmp);
314       }
315       csub(denom, ZERO(in, nu), ZERO(in, i));
316       cdiv(numer, numer, denom);
317       cadd(sum, sum, numer);
318     }
319     ccopy(out, sum);
320     cfree(ctmp);
321     cfree(denom);
322     cfree(numer);
323     rfree(rtmp);
324     cfree(sum);
325   }
326
327   /*
328    * Compute the value of T2(z_nu)
329    */
330   void T2(Complex out, const Zero in, const int nu) {
331     Complex denom;
332     int     i;
333     Complex numer;
334     Complex sum;
335     cnew(denom, 0, 0);
336     cnew(numer, 0, 0);
337     cnew(sum, 0, 0);
338     for (i = 0; i < ZEROS(in); i++) {
339       if (i == nu) {
340         continue;
341       }
342       cset(numer, MULT(in, i), 0);
343       csub(denom, ZERO(in, nu), ZERO(in, i));
344       cmul(denom, denom, denom);
345       cdiv(numer, numer, denom);
346       cadd(sum, sum, numer);
347     }
348     ccopy(out, sum);
349     cfree(denom);
350     cfree(numer);
351     cfree(sum);
```

```
352   }
353
354   /*
355    * Compute the value of T3(z_nu)
356    */
357   void T3(Complex out, const Zero in, const int nu) {
358     Complex ctemp;
359     Complex denom;
360     int     i;
361     Complex numer;
362     Complex sum;
363     cnew(ctemp, 0, 0);
364     cnew(denom, 0, 0);
365     cnew(numer, 0, 0);
366     cnew(sum, 0, 0);
367     for (i = 0; i < ZEROS(in); i++) {
368       if (i == nu) {
369         continue;
370       }
371       cset(numer, MULT(in, i), 0);
372       csub(ctemp, ZERO(in, nu), ZERO(in, i));
373       cmul(denom, ctemp, ctemp);
374       cmul(denom, denom, ctemp);
375       cdiv(numer, numer, denom);
376       cadd(sum, sum, numer);
377     }
378     ccopy(out, sum);
379     cfree(ctemp);
380     cfree(denom);
381     cfree(numer);
382     cfree(sum);
383   }
384
385   /*
386    * Compute the value of u(z)
387    */
388   void u(Complex out, const Complex in) {
389     Complex denom;
390     Complex numer;
391     cnew(denom, 0, 0);
392     cnew(numer, 0, 0);
393     peval(numer, myp, in);
394     peval(denom, myp1, in);
395     /* Search stage, no; iterative stage, yes. */
396     if (count >= 0) {
397       ciota(numer);
398     }
399     cdiv(out, numer, denom);
400     if (debug & 1) {
401       printf("Debug1: Polynomial evaluation (p%d)\nInput\n", precision);
```

```
402       cdump(in);
403       printf("\nNumer\n");
404       cdump(numer);
405       printf("\nDenom\n");
406       cdump(denom);
407       printf("\nOutput\n");
408       cdump(out);
409       printf("\n");
410      }
411     cfree(denom);
412     cfree(numer);
413   }
414
415   /*
416    * Compute the value of u'(z)
417    */
418   void u1(Complex out, const Complex in) {
419     Complex ctmp1;
420     Complex ctmp2;
421     cnew(ctmp1, 2, 0);
422     cnew(ctmp2, 0, 0);
423     A2(ctmp2, in);
424     cmul(ctmp1, ctmp1, ctmp2);
425     u(ctmp2, in);
426     cmul(ctmp1, ctmp1, ctmp2);
427     cset(out, 1, 0);
428     csub(out, out, ctmp1);
429     cfree(ctmp1);
430     cfree(ctmp2);
431   }
```

C.3.7 poly.c

This program file contains the definitions of the functions for manipulating `Poly` values. They obviously rely on the `Real` functions described in §C.3.9, starting on page 161, and the `Complex` functions described in §C.3.1, starting on page 134.

```
1   #include <assert.h>
2   #include <stdio.h>
3   #include "mylib.h"
4   #include "poly.h"
5
6   Poly myp;  /* Global polynomial */
7   Poly myp1; /* First derivative */
8   Poly myp2; /* Second derivative */
9   Poly myp3; /* Third derivative */
```

```
10
11  /*
12   * Add together two Polys, assuming their degrees are
13   * the same (for Eulerian polynomial)
14   */
15  void padd(Poly out, const Poly lhs, const Poly rhs) {
16    int i;
17    assert(PSIZE(out) > DEGREE(lhs));
18    DEGREE(out) = DEGREE(lhs);
19    for (i = 0; i <= DEGREE(out); i++) {
20      cadd(COEFF(out, i), COEFF(lhs, i), COEFF(rhs, i));
21    }
22  }
23
24  /*
25   * Take a copy of a Poly, ensuring sufficient space
26   */
27  void pcopy(Poly out, const Poly in) {
28    int i;
29    assert(PSIZE(out) > DEGREE(in));
30    DEGREE(out) = DEGREE(in);
31    for (i = DEGREE(out); i >= 0; i--) {
32      ccopy(COEFF(out, i), COEFF(in, i));
33    }
34  }
35
36  /*
37   * Compute the degree of a Poly
38   */
39  int pdegree(const Poly in) {
40    int  i;
41    Real tmp;
42    rnew(tmp, 0);
43    for (i = DEGREE(in); i > 0; i--) {
44      cabs(tmp, COEFF(in, i));
45      if (rcmpd(tmp, REAL_EPSILON) > 0) {
46        break;
47      }
48    }
49    rfree(tmp);
50    return i;
51  }
52
53  /*
54   * Compute the coefficients of the differential of a
55   * Poly, allocating sufficient space
56   */
57  void pdiff(Poly out, const Poly in) {
58    int i;
59    if (DEGREE(in) == 0) {
```

```
60        pnew(out, 0);
61      } else {
62        pnew(out, DEGREE(in)-1);
63        for (i = DEGREE(out); i >= 0; i--) {
64          cimul(COEFF(out, i), i+1, COEFF(in, i+1));
65        }
66      }
67    }
68
69    /*
70     * Divide a Poly by an integer
71     */
72    void pdivi(Poly out, const Poly numer, const int denom) {
73      Real tmp;
74      rnew(tmp, denom);
75      pdivr(out, numer, tmp);
76      rfree(tmp);
77    }
78
79    /*
80     * Divide a Poly by a Real number
81     */
82    void pdivr(Poly out, const Poly numer, const Real denom) {
83      int  i;
84      assert(PSIZE(out) > DEGREE(numer));
85      DEGREE(out) = DEGREE(numer);
86      for (i = 0; i <= DEGREE(out); i++) {
87        cdivr(COEFF(out, i), COEFF(numer, i), denom);
88      }
89    }
90
91    /*
92     * Dump a Poly to the highest possible accurancy
93     */
94    void pdump(const Poly in) {
95      int i;
96      printf("%d\n", DEGREE(in));
97      for (i = DEGREE(in); i >= 0; i--) {
98        cdump(COEFF(in, i));
99        printf("\n");
100     }
101   }
102
103   /*
104    * Evaluate a Poly
105    */
106   void peval(Complex out, const Poly in, const Complex c) {
107     int i;
108     cnew(out, 0, 0);
109     for (i = DEGREE(in); i >= 0; i--) {
```

```
110       cmul(out, out, c);
111       cadd(out, out, COEFF(in, i));
112     }
113   }
114
115   /*
116    * Free the space allocated for a Poly
117    */
118   void pfree(Poly in) {
119     int i;
120     for (i = 0; i < PSIZE(in); i++) {
121       cfree(COEFF(in, i));
122     }
123     free(in->coeff);
124   }
125
126   /*
127    * Multiply an integer by a Poly
128    */
129   void pimul(Poly out, const int n, const Poly in) {
130     int  i;
131     assert(PSIZE(out) > DEGREE(in));
132     DEGREE(out) = DEGREE(in);
133     for (i = 0; i <= DEGREE(out); i++) {
134       cimul(COEFF(out, i), n, COEFF(in, i));
135     }
136   }
137
138   /*
139    * Input a Poly, allocating sufficient space
140    */
141   void pinput(Poly out) {
142     int i;
143     DEGREE(out) = getinteger();
144     PSIZE(out) = DEGREE(out)+1;
145     if ((out->coeff = calloc(PSIZE(out), sizeof(Complex))) == NULL) {
146       fatal("Pinput out of memory %d", PSIZE(out));
147     }
148     for (i = DEGREE(out); i >= 0; i--) {
149       cinput(COEFF(out, i));
150     }
151   }
152
153   /*
154    * Put a Poly into monic form
155    */
156   void pmonic(Poly out, const Poly in) {
157     int i;
158     pcopy(out, in);
159     for (i = 0; i < DEGREE(out); i++) {
```

```
160       cdiv(COEFF(out, i), COEFF(out, i), COEFF(out, DEGREE(out)));
161     }
162     cset(COEFF(out, DEGREE(out)), 1, 0);
163   }
164
165   /*
166    * Multiply two Polys
167    */
168   void pmul(Poly out, const Poly lhs, const Poly rhs) {
169     Complex ctmp;
170     int     i;
171     int     j;
172     Poly    ptmp;
173     cnew(ctmp, 0, 0);
174     pnew(ptmp, DEGREE(lhs)+DEGREE(rhs));
175     for (i = 0; i <= DEGREE(lhs); i++) {
176       for (j = 0; j <= DEGREE(rhs); j++) {
177         cmul(ctmp, COEFF(lhs, i), COEFF(rhs, j));
178         cadd(COEFF(ptmp, i+j), COEFF(ptmp, i+j), ctmp);
179       }
180     }
181     pcopy(out, ptmp);
182     cfree(ctmp);
183     pfree(ptmp);
184   }
185
186   /*
187    * Allocate space for a Poly
188    */
189   void pnew(Poly out, const int degree) {
190     int i;
191     DEGREE(out) = degree;
192     PSIZE(out) = DEGREE(out)+1;
193     if ((out->coeff = calloc(PSIZE(out), sizeof(Complex))) == NULL) {
194       fatal("Pnew out of memory %d", PSIZE(out));
195     }
196     for (i = 0; i < PSIZE(out); i++) {
197       cnew(COEFF(out, i), 0, 0);
198     }
199   }
200
201   /*
202    * Output a Poly
203    */
204   void poutput(const Poly in) {
205     int i;
206     for (i = DEGREE(in); i >= 0; i--) {
207       printf("  p[%2d] ", i);
208       coutput(COEFF(in, i));
209       printf("\n");
```

```
210     }
211     fflush(stdout);
212   }
213
214   /*
215    * Scale a Poly, out(z) = in(radius*z), no special
216    * steps, see Stewart (1969)
217    */
218   void pscale(Poly out, const Poly in, const double radius) {
219     int  i;
220     Real rtmp;
221     Real rho;
222     rnew(rho, radius);
223     rnew(rtmp, 1);
224     assert(PSIZE(out) > DEGREE(in));
225     DEGREE(out) = DEGREE(in);
226     for (i = 0; i <= DEGREE(out); i++) {
227       crmul(COEFF(out, i), rtmp, COEFF(in, i));
228       rmul(rtmp, rtmp, rho);
229     }
230     if (debug == 4) {
231       printf("Pscale\n");
232       poutput(out);
233     }
234     rfree(rtmp);
235     rfree(rho);
236   }
237
238   /*
239    * Scale a Poly to monic form and remove zeros at the
240    * origin
241    */
242   void psetup(Poly out, const Poly in) {
243     int  i;
244     int  n;
245     Real rtmp;
246     rnew(rtmp, 0);
247     pcopy(out, in);
248     for (n = 0; n <= DEGREE(out); n++) {
249       cabs(rtmp, COEFF(out, n));
250       if (rget(rtmp) > 0) {
251         break;
252       }
253     }
254     if (n > 0) {
255       printf("Removing %d zeros at the origin\n", n);
256       DEGREE(out) -= n;
257       for (i = 0; i <= DEGREE(out); i++) {
258         ccopy(COEFF(out, i), COEFF(out, n+i));
259       }
```

```
260     }
261     pmonic(out, out);
262     rfree(rtmp);
263   }
264
265   /*
266    * Shift a Poly, out(z) = in(z+c), see Stewart (1969)
267    */
268   void pshift(Poly out, const Poly in, const Complex c) {
269     Complex ctmp;
270     int     i;
271     cnew(ctmp, 0, 0);
272     pcopy(out, in);
273     for (i = 1; i <= DEGREE(out); i++) {
274       int j;
275       for (j = DEGREE(out)-i; j < DEGREE(out); j++) {
276         cmul(ctmp, COEFF(out, j+1), c);
277         cadd(COEFF(out, j), COEFF(out, j), ctmp);
278       }
279     }
280     cfree(ctmp);
281   }
282
283   /*
284    * Smooth a Poly, see Marden for reasoning
285    */
286   void psmooth(Poly out, const Poly in) {
287     Real raverage;
288     int  i;
289     Real rtmp;
290     rnew(raverage, 0);
291     rnew(rtmp, 0);
292     for (i = 0; i <= DEGREE(in); i++) {
293       cabs(rtmp, COEFF(in, i));
294       radd(raverage, raverage, rtmp);
295     }
296     rset(rtmp, DEGREE(in)+1);
297     rdiv(raverage, raverage, rtmp);
298     pdivr(out, in, raverage);
299     rfree(raverage);
300     rfree(rtmp);
301   }
302
303   /*
304    * Subtract two Polys (used to build Laguerre polynomials)
305    */
306   void psub(Poly out, const Poly lhs, const Poly rhs) {
307     int i;
308     if (DEGREE(lhs) > DEGREE(rhs)) {
309       pcopy(out, lhs);
```

```
310       for (i = 0; i <= DEGREE(rhs); i++) {
311         csub(COEFF(out, i), COEFF(out, i), COEFF(rhs, i));
312       }
313     } else {
314       pcopy(out, rhs);
315       for (i = 0; i <= DEGREE(lhs); i++) {
316         csub(COEFF(out, i), COEFF(out, i), COEFF(lhs, i));
317       }
318     }
319   }
```

C.3.8 rall2.c

This program file contains the function for evaluating Louis Rall's second-order IF defined in Equation (B.9) on page 122.

```
1   #include "mylib.h"
2   #include "rall2.h"
3   #include "zero.h"
4
5   /*
6    * Main program for Rall's second-order IF
7    */
8   int main(int argc, char *argv[]) {
9     return run(argc, argv, &rall2, "Rall's second-order");
10  }
11
12  /*
13   * Perform one iteration of Rall's method
14   */
15  void rall2(Zero out, const Zero in, const char mode) {
16    int  nu;
17    Zero z;
18    znew(z, ZEROS(in));
19    zcopy(z, in);
20    for (nu = 0; nu < ZEROS(z); nu++) {
21      if (done[nu]) {
22        continue;
23      }
24      newton2(ZERO(out, nu), ZERO(z, nu), MULT(z, nu));
25    }
26    zfree(z);
27  }
```

C.3.9 real.c

This program file contains the definitions of the functions for manipulating `Real` values. The majority of these use the low-level functions defined in the GNU Multiple Precision Arithmetic Library (GMP) [Gra11].

```
1  #include <stdio.h>
2  #include "mylib.h"
3  #include "real.h"
4
5  /*
6   * Absolute value of a Real
7   */
8  void rabs(Real out, const Real in) {
9    mpf_abs(out, in);
10 }
11
12 /*
13  * Add two Reals
14  */
15 void radd(Real out, const Real lhs, const Real rhs) {
16   mpf_add(out, lhs, rhs);
17 }
18
19 /*
20  * Compare two Reals (< is -1, == is 0, > is 1)
21  */
22 int rcmp(const Real lhs, const Real rhs) {
23   return mpf_cmp(lhs, rhs);
24 }
25
26 /*
27  * Compare a Real and a double (< is -1, == is 0, > is 1)
28  */
29 int rcmpd(const Real lhs, const double rhs) {
30   return mpf_cmp_d(lhs, rhs);
31 }
32
33 /*
34  * Copy a Real
35  */
36 void rcopy(Real out, const Real in) {
37   mpf_set(out, in);
38 }
39
40 /*
41  * Divide two Reals
42  */
```

```
43  void rdiv(Real out, const Real numer, const Real denom) {
44    mpf_div(out, numer, denom);
45  }
46
47  /*
48   * Dump a Real to the highest possible accurancy
49   */
50  void rdump(const Real in) {
51    gmp_printf("% .Fe", in);
52  }
53
54  /*
55   * Free the storage used for a Real
56   */
57  void rfree(Real in) {
58    mpf_clear(in);
59  }
60
61  /*
62   * Get the double precision value of a Real
63   */
64  double rget(const Real in) {
65    return mpf_get_d(in);
66  }
67
68  /*
69   * Initialise storage requirements for Reals
70   */
71  int rinit(const int new) {
72    int old = mpf_get_default_prec();
73    mpf_set_default_prec(new);
74    printf("Precision %d\n", new);
75    return old;
76  }
77
78  /*
79   * Input a Real
80   */
81  void rinput(Real out) {
82    mpf_init(out);
83    if (gmp_scanf("%Fe", out) != 1) {
84      fatal("Real input error", NULL);
85    }
86  }
87
88  /*
89   * Set a very small Real to zero
90   */
91  void riota(Real r) {
92    Real tmp;
```

```
93      rnew(tmp, 0);
94      rabs(tmp, r);
95      if (rcmpd(tmp, REAL_IOTA) < 0) {
96        rset(r, 0);
97      }
98      rfree(tmp);
99    }
100
101   /*
102    * Multiply two Reals
103    */
104   void rmul(Real out, const Real lhs, const Real rhs) {
105     mpf_mul(out, lhs, rhs);
106   }
107
108   /*
109    * Negate a Real
110    */
111   void rneg(Real out, const Real in) {
112     mpf_neg(out, in);
113   }
114
115   /*
116    * Create a new Real from a double precision value
117    */
118   void rnew(Real out, const double in) {
119     mpf_init_set_d(out, in);
120   }
121
122   /*
123    * Output a Real
124    */
125   void routput(const Real in) {
126     gmp_printf(REAL_FORMAT, in);
127   }
128
129   /*
130    * Convert a double precision into a Real
131    */
132   void rset(Real out, const double in) {
133     mpf_set_d(out, in);
134   }
135
136   /*
137    * Compute the square root of a Real
138    */
139   void rsqrt(Real out, const Real in) {
140     mpf_sqrt(out, in);
141   }
142
```

```
143  /*
144   * Subtract two Reals
145   */
146  void rsub(Real out, const Real lhs, const Real rhs) {
147    mpf_sub(out, lhs, rhs);
148  }
149
150  /*
151   * swap two Real numbers
152   */
153  void rswap(Real lhs, Real rhs) {
154    mpf_swap(lhs, rhs);
155  }
```

C.3.10 search.c

This program contains the definitions of our workhorse search function together with a number of auxiliary functions that are required during the search stage.

```
1   #include <assert.h>
2   #include <getopt.h>
3   #include <math.h>
4   #include <signal.h>
5   #include <time.h>
6   #include "mylib.h"
7   #include "poly.h"
8   #include "search.h"
9   #include "zero.h"
10
11  /*
12   * Locate a disk about the origin containing all the zeros
13   * of the polynomial p.
14   */
15  double dekker(const Poly p) {
16    int    i;
17    double max = 0;
18    Real   r;
19    rnew(r, 0);
20    for (i = 0; i <= DEGREE(p); i++) {
21      double d;
22      cabs(r, COEFF(p, i));
23      d = pow(rget(r), 1.0/(DEGREE(p)-i));
24      if (d > max) {
25        max = d;
26      }
27    }
28    rfree(r);
```

```
29      return 2*max;
30    }
31
32    /*
33     * The limit of 1/u1(z) tends to the multiplicity
34     * of the approximation of the zero in.
35     * Lagouanelle (1966).
36     */
37    int lagouanelle(const Complex in) {
38      Complex c;
39      int     m;
40      Real    r;
41      cnew(c, 0, 0);
42      rnew(r, 0);
43      u1(c, in);
44      cabs(r, c);
45      m = 1/rget(r)+0.5;
46      cfree(c);
47      rfree(r);
48      return m;
49    }
50
51    /*
52     * Main program for the search phase
53     */
54    int main(int argc, char *argv[]) {
55      int    c;
56      double cover = COVER;
57      int    max1  = MAX1;
58      int    max2  = MAX2;
59      int    sides = SIDES;
60      int    size;
61      Real   rtmp;
62      Zero   z;
63      int    zeros = 0;
64      signal(SIGABRT, sighandler);
65      rnew(rtmp, 0);
66      while ((c = getopt(argc, argv, "c:d:m:n:p:s:z:")) != -1) {
67        switch (c) {
68        case 'c':
69          cover = strtod(optarg, NULL);
70          break;
71        case 'd':
72          debug = atoi(optarg);
73          break;
74        case 'm':
75          max1 = atoi(optarg);
76          break;
77        case 'n':
78          max2 = atoi(optarg);
```

```
79          break;
80        case 'p':
81          precision = atoi(optarg);
82          break;
83        case 's':
84          sides = atoi(optarg);
85          break;
86        case 'z':
87          zeros = atoi(optarg);
88          break;
89        default:
90          fatal("Unknown option %s", argv[optind-1]);
91        }
92      }
93      rinit(precision);
94      printf("Search the complex plane sides %d cover %g\n",
95             sides, cover);
96      printf("Max outer iterations %d", max1);
97      printf(" Max inner iterations %d", max2);
98      printf("\n");
99      pinput(myp);
100     printf("Original polynomial\n");
101     poutput(myp);
102     cabs(rtmp, COEFF(myp, DEGREE(myp)));
103     if (rcmpd(rtmp, REAL_EPSILON) < 0) {
104       error("Leading coefficient too small");
105       zfree(z);
106       return(EXIT_FAILURE);
107     }
108     psetup(myp, myp);
109     printf("Monic polynomial\n");
110     poutput(myp);
111     if (DEGREE(myp) < 2) {
112       error("Degenerate polynomial\n");
113       zfree(z);
114       return(EXIT_FAILURE);
115     }
116     pdiff(myp1, myp);
117     pdiff(myp2, myp1);
118     if (zeros) {
119       size = zeros;
120     } else {
121       size = 6*DEGREE(myp);
122     }
123     znew(z, size);
124     if (search(z, myp, max1, max2, sides, cover)) {
125       return(EXIT_FAILURE);
126     }
127     zdump(z);
128     pfree(myp);
```

```
129     pfree(myp1);
130     pfree(myp2);
131     rfree(rtmp);
132     zfree(z);
133     return(EXIT_SUCCESS);
134   }
135
136   /*
137    * Count the number of zeros of polynomial p in the disk (centre, radius).
138    * A variation on Marden (1947).
139    */
140   int marden(const Poly p, const Complex centre, const double radius) {
141     int  count = 0;
142     int  i;
143     int  sign = 1;
144     Poly tmp;
145     pnew(tmp, DEGREE(p));
146     pshift(tmp, p, centre);
147     pscale(tmp, tmp, radius);
148     for (i = 0; i < DEGREE(p); i++) {
149       double delta;
150       reduce(tmp, tmp);
151       /*
152        * Attempt to avoid delta values being zero.
153        * See Marden paper.
154        */
155       if (DEGREE(tmp) > 0) {
156         psmooth(tmp, tmp);
157       }
158       delta = rget(REAL(COEFF(tmp, 0)));
159       /*
160       if (fabs(delta) < 0.01*REAL_EPSILON) {
161         error("Marden delta %g", delta);
162         poutput(tmp);
163         break;
164       }
165       */
166       if (sign ^ (delta > 0)) {
167         sign = 0;
168         count++;
169       } else {
170         sign = 1;
171       }
172     }
173     pfree(tmp);
174     return count;
175   }
176
177   /*
178    * Pave the square (centre, radius) with n^2 squares each with
```

```
179   * radius radius/n, test each square for a zero and, if so,
180   * add it to the set of zeros z.  Return the new radius.
181   */
182  double pave(Zero z, const Poly p, const Complex centre,
183              const double radius, const int n) {
184    Complex c;
185    int     i;
186    double  r = radius/n;
187    Complex start;
188    cnew(c, 0, 0);
189    cnew(start, 0, 0);
190    for (i = 0; i < n; i++) {
191      int j;
192      cset(c, -i*r, (n-1-i)*r);
193      cadd(start, centre, c);
194      for (j = 0; j < n; j++) {
195        cset(c, j*r, -j*r);
196        cadd(c, start, c);
197        test(z, p, c, r);
198        /*
199        printf("Cover %d %d %d ", i, j, ZEROS(z));
200        coutput(c);
201        printf("\n");
202        */
203      }
204    }
205    cfree(c);
206    cfree(start);
207    return r;
208  }
209
210  /*
211   * Reduce the degree of the polynomial (by one?).
212   * See the Marden paper.
213   */
214  void reduce(Poly out, const Poly in) {
215    Complex ctmp1;
216    Complex ctmp2;
217    Complex ctmp3;
218    int     i;
219    Poly    ptmp;
220    assert(PSIZE(out) >= DEGREE(in));
221    cnew(ctmp1, 0, 0);
222    cnew(ctmp2, 0, 0);
223    cnew(ctmp3, 0, 0);
224    pnew(ptmp, DEGREE(in));
225    pcopy(ptmp, in);
226    DEGREE(out) = DEGREE(ptmp)-1;
227    cconj(ctmp1, COEFF(ptmp, 0));
228    for (i = 0; i <= DEGREE(out); i++) {
```

```
229       cmul(ctmp2, ctmp1, COEFF(ptmp, i));
230       cconj(ctmp3, COEFF(ptmp, DEGREE(ptmp)-i));
231       cmul(ctmp3, COEFF(ptmp, DEGREE(ptmp)), ctmp3);
232       csub(COEFF(out, i), ctmp2, ctmp3);
233     }
234     cfree(ctmp1);
235     cfree(ctmp2);
236     cfree(ctmp3);
237     pfree(ptmp);
238   }
239
240   /*
241    * Search the complex plane for squares containing zeros of
242    * the polynomial p.
243    */
244   int search(Zero z, const Poly p, const int max1, const int max2,
245              const int sides, const double cover) {
246     int     agree;
247     int     iter1;
248     double  radius = dekker(p);
249     double  seconds;
250     clock_t tstart = clock();
251     Zero    ztmp;
252     printf("Search dekker radius %g\n", radius);
253     radius *= sqrt(2);
254     printf("Search semi-diagonal (radius) %g\n", radius);
255     znew(ztmp, ZSIZE(z));
256     cset(ZERO(ztmp, 0), 0, 0);
257     ZEROS(ztmp) = 1;
258     for (iter1 = 1; iter1 <= max1; iter1++) {
259       int i;
260       int iter2;
261       printf("Search outer iteration %d (of %d) disks %d\n",
262              iter1, max1, ZEROS(ztmp));
263       for (iter2 = 1; iter2 <= max2; iter2++) {
264         double r;
265         printf("Search inner iteration %d (of %d) disks %d radius %g\n",
266                iter2, max2, ZEROS(ztmp), radius);
267         ZEROS(z) = 0;
268         for (i = 0; i < ZEROS(ztmp); i++) {
269           r = pave(z, p, ZERO(ztmp, i), radius, sides);
270         }
271         if (ZEROS(z) == 0) {
272           fatal("Search lost zeros!");
273         }
274         radius = r;
275         zcopy(ztmp, z);
276       }
277       printf("Search inner done\n");
278       zcover(z, ztmp, radius, cover*radius);
```

```
279       radius *= cover;
280       for (i = 0; i < ZEROS(z); i++) {
281         MULT(z, i) = marden(p, ZERO(z, i), radius);
282       }
283       printf("Search zeros reduced from %d to %d radius %g\n",
284              ZEROS(ztmp), ZEROS(z), radius);
285       zoutput(z);
286       agree = 1;
287       for (i = 0; i < ZEROS(z); i++) {
288         int lag = lagouanelle(ZERO(z, i));
289         printf("Search lagouanelle[%2d] %d\n", i+1, lag);
290         if (lag != MULT(z, i)) {
291           agree = 0;
292         }
293       }
294       if (zdegree(z) < DEGREE(p)) {
295         error("Search missing zeros %d", zdegree(z));
296         break;
297       }
298       zcopy(ztmp, z);
299     }
300     zfree(ztmp);
301     printf("SEARCH TIME %f\n",
302            (double)(clock()-tstart)/CLOCKS_PER_SEC);
303     if (agree && zdegree(z) == DEGREE(p)) {
304       FILE *fd;
305       if ((fd = fopen("/tmp/iters", "w")) == NULL) {
306         fatal("Iterate can't open temporary file");
307       }
308       fprintf(fd, "%d\n", iter1-1);
309       fclose(fd);
310       printf("Search success\n");
311       return(EXIT_SUCCESS);
312     }
313     printf("Search failure\n");
314     return(EXIT_FAILURE);
315   }
316
317   /*
318    * Test whether polynomial p has a zero in the disk
319    * (centre, radius), if so adding it to the set of
320    * zeros z.
321    */
322   void test(Zero z, const Poly p, const Complex centre,
323             const double radius) {
324     int m;
325     if ((m = marden(p, centre, radius)) == 0) {
326       return;
327     }
328     assert(ZSIZE(z) > ZEROS(z));
```

```
329     ccopy(ZERO(z, ZEROS(z)), centre);
330     MULT(z, ZEROS(z)) = m;
331     ZEROS(z)++;
332   }
```

C.3.11 traub3.c

This program file contains the function for evaluating Joseph Traub's third-order IF defined in Equation (5.37) on page 58.

```
1   #include <stdio.h>
2   #include "mylib.h"
3   #include "traub3.h"
4   #include "zero.h"
5
6   int main(int argc, char *argv[]) {
7     return run(argc, argv, &traub3, "Traub's third-order (multiple zeros)");
8   }
9
10  /*
11   * Perform one iteration of Traub's third-order multiple
12   * zero method (p. 139)
13   */
14  void traub3(Zero out, const Zero in, const char mode) {
15    Complex ctmp;
16    Complex ctmp2;
17    Complex ctmp3;
18    int     nu;
19    Complex utmp;
20    Zero    z;
21    cnew(ctmp, 0, 0);
22    cnew(ctmp2, 0, 0);
23    cnew(ctmp3, 0, 0);
24    cnew(utmp, 0, 0);
25    znew(z, ZEROS(in));
26    zcopy(z, in);
27    for (nu = 0; nu < ZEROS(z); nu++) {
28      if (done[nu]) {
29        continue;
30      }
31      u(utmp, ZERO(z, nu));
32      cimul(ctmp, MULT(z, nu), utmp);
33      A2(ctmp2, ZERO(z, nu));
34      cmul(ctmp2, ctmp2, utmp);
35      cimul(ctmp2, MULT(z, nu), ctmp2);
36      cset(ctmp3, 0.5*(3-MULT(z, nu)), 0);
37      cadd(ctmp2, ctmp3, ctmp2);
```

```
38        cmul(ctmp, ctmp, ctmp2);
39        csub(ZERO(out, nu), ZERO(z, nu), ctmp);
40      }
41      cfree(ctmp);
42      cfree(ctmp2);
43      cfree(ctmp3);
44      cfree(utmp);
45      zfree(z);
46    }
```

C.3.12 zero.c

This program file contains the definitions of the functions that manipulate vectors of `Zero` values. These functions rely on the functions for `Real` values, see §C.3.9 starting on page 161, `Complex` values, see §C.3.1 starting on page 134, and `Poly` values, see §C.3.7 starting on page 153.

```
1   #include <assert.h>
2   #include "complex.h"
3   #include "mylib.h"
4   #include "zero.h"
5
6   /*
7    * Copy a set of zeros
8    */
9   void zcopy(Zero out, const Zero in) {
10    int i;
11    assert(ZSIZE(out) >= ZEROS(in));
12    ZEROS(out) = ZEROS(in);
13    for (i = 0; i < ZEROS(in); i++) {
14      ccopy(ZERO(out, i), ZERO(in, i));
15      MULT(out, i) = MULT(in, i);
16    }
17  }
18
19  /* Remove disks of radius oldrad wholly covered by other
20   * disks of radius newrad
21   */
22  void zcover(Zero out, const Zero in, const double oldrad,
23              const double newrad) {
24    Complex ctmp;
25    int     i;
26    Real    rtmp;
27    cnew(ctmp, 0, 0);
28    rnew(rtmp, 0);
29    /* Candidates for remaining */
30    for (i = 0; i < ZEROS(in); i++) {
```

```
31        int j;
32        if (MULT(in, i) == 0) {
33          continue;
34        }
35        /* Candidates for removal */
36        for (j = i+1; j < ZEROS(in); j++) {
37          double dtmp;
38          if (MULT(in, j) == 0) {
39            continue;
40          }
41          csub(ctmp, ZERO(in, i), ZERO(in, j));
42          cabs(rtmp, ctmp);
43          dtmp = rget(rtmp)+oldrad;
44          /*
45          printf("%d %d\n", i, j);
46          coutput(ZERO(in, i));
47          coutput(ZERO(in, j));
48          printf(" dtmp %g newrad %g\n", dtmp, newrad);
49          */
50          if (dtmp < newrad) {
51            MULT(in, j) = 0;
52          }
53        }
54      }
55      ZEROS(out) = 0;
56      for (i = 0; i < ZEROS(in); i++) {
57        if (MULT(in, i) == 0) {
58          continue;
59        }
60        assert(ZSIZE(out) > ZEROS(out));
61        ccopy(ZERO(out, ZEROS(out)), ZERO(in, i));
62        MULT(out, ZEROS(out)) = MULT(in, i);
63        ZEROS(out)++;
64      }
65      cfree(ctmp);
66      rfree(rtmp);
67    }
68
69    /*
70     * Compute the polynomial degree from a set of zeros
71     */
72    int zdegree(const Zero in) {
73      int count = 0;
74      int i;
75      for (i = 0; i < ZEROS(in); i++) {
76        count += MULT(in, i);
77      }
78      return count;
79    }
80
```

```
81  /*
82   * Dump a set of zeros to greatest possible accuracy
83   */
84  void zdump(const Zero in) {
85    int   i;
86    FILE *fd;
87    if ((fd = fopen("/tmp/zeros", "w")) == NULL) {
88      fatal("Zdump can't open temporary file");
89    }
90    fprintf(fd, "%d\n", ZEROS(in));
91    for (i = 0; i < ZEROS(in); i++) {
92      gmp_fprintf(fd, "% .Fe", REAL(ZERO(in, i)));
93      fprintf(fd, " ");
94      gmp_fprintf(fd, "% .Fe", IMAG(ZERO(in, i)));
95      fprintf(fd, " %d\n", MULT(in, i));
96    }
97    fclose(fd);
98  }
99
100 /*
101  * Free the storage used by a set of zeros
102  */
103 void zfree(Zero in) {
104   int i;
105   free(in->mult);
106   for (i = 0; i < ZSIZE(in); i++) {
107     cfree(ZERO(in, i));
108   }
109   free(in->zero);
110 }
111
112 /*
113  * Input a set of zeros
114  */
115 void zinput(Zero out) {
116   int i;
117   ZEROS(out) = ZSIZE(out) = getinteger();
118   if ((out->zero = calloc(ZEROS(out), sizeof(Complex))) == NULL) {
119     fatal("Zinput out of memory (zero) %d", ZSIZE(out));
120   }
121   if ((out->mult = calloc(ZEROS(out), sizeof(int))) == NULL) {
122     fatal("Zinput out of memory (mult) %d", ZSIZE(out));
123   }
124   for (i = 0; i < ZEROS(out); i++) {
125     cinput(ZERO(out, i));
126     MULT(out, i) = iinput();
127   }
128 }
129
130 /*
```

```
131   * Allocate storage for a new set of zeros.
132   */
133  void znew(Zero out, const int count) {
134    int i;
135    ZEROS(out) = ZSIZE(out) = count;
136    if ((out->zero = calloc(ZEROS(out), sizeof(Complex))) == NULL) {
137      fatal("Znew out of memory (zero) %d", ZSIZE(out));
138    }
139    if ((out->mult = calloc(ZEROS(out), sizeof(int))) == NULL) {
140      fatal("Znew out of memory (mult) %d", ZSIZE(out));
141    }
142    /* Do we need this? */
143    for (i = 0; i < ZEROS(out); i++) {
144      cnew(ZERO(out, i), 0, 0);
145      MULT(out, i) = 0;
146    }
147  }
148
149  /*
150   * Set very small Zeros to zero
151   */
152  void ziota(Zero z) {
153    int i;
154    for (i = 0; i < ZEROS(z); i++) {
155      ciota(ZERO(z, i));
156    }
157  }
158
159  /*
160   * Output a set of zeros
161   */
162  void zoutput(const Zero in) {
163    int i;
164    for (i = 0; i < ZEROS(in); i++) {
165      printf("  z[%2d] ", i+1);
166      coutput(ZERO(in, i));
167      printf(" %2d\n", MULT(in, i));
168    }
169    fflush(stdout);
170  }
171
172  /*
173   * Partition the zeros around a pivot value
174   */
175  int zpartition(Zero in, const int lhs, const int rhs, int index) {
176    int i;
177    Complex pivot;
178    cnew(pivot, 0, 0);
179    ccopy(pivot, ZERO(in, index));
180    zswap(in, index, rhs);
```

```
181     index = lhs;
182     for (i = lhs; i < rhs; i++) {
183       if (ccmp(ZERO(in, i), pivot) <= 0) {
184         zswap(in, i, index++);
185       }
186     }
187     zswap(in, index, rhs);
188     cfree(pivot);
189     return index;
190   }
191
192   /*
193    * Sort an array of zeros in-place by increasing complex
194    * centres
195    */
196   void zsort(Zero in, const int lhs, const int rhs) {
197     if (rhs > lhs) {
198       int index = zpartition(in, lhs, rhs, (lhs+rhs)/2);
199       zsort(in, lhs, index-1);
200       zsort(in, index+1, rhs);
201     }
202   }
203
204   /*
205    * Swap two zeros in an array of zeros
206    */
207   void zswap(Zero in, const int lhs, const int rhs) {
208     int tmp = MULT(in, lhs);
209     cswap(ZERO(in, lhs), ZERO(in, rhs));
210     MULT(in, lhs) = MULT(in, rhs);
211     MULT(in, rhs) = tmp;
212   }
```

C.4 C Auxiliary Program Files

This section describes a number of C programs that we use to construct polynomial coefficients, i.e. the vital data required during both stages of our algoritm, from a variety of different sources. Such sources can be as varied as recurrence relations for generating the coefficients through a simple list of exact zeros from a reliable source.

C.4.1 ebuild.c

This program builds the coefficients of an Eulerian polynomial using the recurrence relation given in [MW08, p. 1]. The zeros are all real.

```
1   /*
2    * Build a Eulerian polynomial
3    */
4   #include <stdio.h>
5   #include <stdlib.h>
6   #include "mylib.h"
7   #include "poly.h"
8   #include "zero.h"
9
10  char *optarg;
11  int   opterr, optind, optopt;
12
13  /*
14   * Main program to build an Eulerian polynomial from the
15   * recurrence relation given in Ma, et.al.
16   */
17  int main(int argc, char *argv[]) {
18    int  c;
19    int  degree = 0;
20    Poly fact0;
21    Poly fact1;
22    int  i;
23    Poly ptmp;
24    while ((c = getopt(argc, argv, "n:p:")) != -1) {
25      switch (c) {
26      case 'n':
27        degree = atoi(optarg);
28        break;
29      case 'p':
30        precision = atoi(optarg);
31        break;
32      default:
33        fatal("Unknown option %s", argv[optind-1]);
34      }
35    }
36    rinit(precision);
37    pnew(fact0, 1);
38    pnew(fact1, 2);
39    pnew(myp, degree);
40    pnew(myp1, degree-1);
41    pnew(ptmp, degree);
42    DEGREE(fact0) = 1;
43    cset(COEFF(fact0, 1), 1, 0);
44    cset(COEFF(fact0, 0), 0, 0);
45    DEGREE(fact1) = 2;
46    cset(COEFF(fact1, 2), -1, 0);
47    cset(COEFF(fact1, 1), 1, 0);
48    cset(COEFF(fact1, 0), 0, 0);
49    DEGREE(myp) = 1;
50    cset(COEFF(myp, 1), 1, 0);
```

```
51      cset(COEFF(myp, 0), 0, 0);
52      for (i = 2; i <= degree; i++) {
53        pdiff(myp1, myp);
54        pimul(myp, i, myp);
55        pmul(myp, fact0, myp);
56        pmul(ptmp, fact1, myp1);
57        padd(myp, myp, ptmp);
58      }
59      printf("# Used ebuild -p%d\n", precision);
60      pdump(myp);
61      pfree(fact0);
62      pfree(fact1);
63      pfree(ptmp);
64    }
```

C.4.2 fbuild.c

This program builds the coefficients of an arbitrary polynomial from a collection of factors, some single and some multiple.

```
1   /*
2    * Build a polynomial (on standout) from underlying factors
3    * (on stdin)
4    */
5   #include <stdio.h>
6   #include <stdlib.h>
7   #include "mylib.h"
8   #include "poly.h"
9   #include "zero.h"
10
11  char *optarg;
12  int   opterr, optind, optopt;
13
14  int main(int argc, char *argv[]) {
15    int  c;
16    int  count = 0;
17    int  degree;
18    int  f;
19    int  factors;
20    Poly ptmp;
21    while ((c = getopt(argc, argv, "p:")) != -1) {
22      switch (c) {
23      case 'p':
24        precision = atoi(optarg);
25        break;
26      default:
27        fatal("Unknown option %s", argv[optind-1]);
```

```
28        }
29      }
30      rinit(precision);
31      degree = getinteger();
32      pnew(myp, degree);
33      DEGREE(myp) = 0;
34      cset(COEFF(myp, 0), 1, 0);
35      factors = getinteger();
36      while (factors-- > 0) {
37        int mult = iinput();
38        pinput(ptmp);
39        count += mult*DEGREE(ptmp);
40        while (mult-- > 0) {
41          pmul(myp, myp, ptmp);
42        }
43        pfree(ptmp);
44      }
45      if (count != degree) {
46        fatal("Fbuild size mismatch %d %d\n", degree, count);
47      }
48      printf("# Used fbuild -p%d\n", precision);
49      pdump(myp);
50    }
```

C.4.3 kbuild.c

This program builds the coefficients of a Kirrinnis polynomial using the equation given by Dario Bini as part of his test suite [BF00b, p. 14].

```
1   /*
2    * Build a Kirrinnis polynomial (on stdout).  See Bini p. 14
3    * for details.
4    */
5   #include "mylib.h"
6   #include "poly.h"
7   #include "zero.h"
8
9   char *optarg;
10  int   opterr, optind, optopt;
11
12  int main(int argc, char *argv[]) {
13    int    c;
14    int    degree;
15    double dtmp;
16    double epsilon = 1.0/4096;
17    int    i;
18    int    n = 10;
```

```
19      Poly   ptmp;
20      while ((c = getopt(argc, argv, "n:p:")) != -1) {
21        switch (c) {
22        case 'n':
23          n = atoi(optarg);
24          break;
25        case 'p':
26          precision = atoi(optarg);
27          break;
28        default:
29          fatal("Unknown option %s", argv[optind-1]);
30        }
31      }
32      rinit(precision);
33      pnew(myp, degree = 4*n+4);
34      DEGREE(myp) = 4;
35      cset(COEFF(myp, 4), 1, 0);
36      dtmp = 0.5+epsilon;
37      cset(COEFF(myp, 0), dtmp*dtmp*dtmp*dtmp, 0);
38      pdump(myp);
39      pnew(ptmp, 4);
40      cset(COEFF(ptmp, 4), 1, 0);
41      cset(COEFF(ptmp, 0), -0.0625, 0);
42      pdump(ptmp);
43      for (i = 1; i <= n; i++) {
44        pmul(myp, myp, ptmp);
45      }
46      printf("# Used kbuild -n%d -p%d\n", n, precision);
47      pdump(myp);
48      pfree(myp);
49      pfree(ptmp);
50    }
```

C.4.4 lbuild.c

This program builds the coefficients of a Laguerre polynomial using the recurrence relation given in [AS70, pp. 799–780].

```
1   /*
2    * Build a Laguerre polynomial
3    */
4   #include <stdio.h>
5   #include <stdlib.h>
6   #include "mylib.h"
7   #include "poly.h"
8   #include "zero.h"
9
```

```
10  char *optarg;
11  int   opterr, optind, optopt;
12
13  /*
14   * Main program to build a Laguerre polynomial from the
15   * recurrence relation given in Abromowitz & Stegun
16   */
17  int main(int argc, char *argv[]) {
18    int  c;
19    int  degree = 0;
20    int  i;
21    Poly lag0;
22    Poly lag1;
23    Poly ptmp;
24    while ((c = getopt(argc, argv, "n:p:")) != -1) {
25      switch (c) {
26      case 'n':
27        degree = atoi(optarg);
28        break;
29      case 'p':
30        precision = atoi(optarg);
31        break;
32      default:
33        fatal("Unknown option %s", argv[optind-1]);
34      }
35    }
36    rinit(precision);
37    pnew(lag0, degree);
38    pnew(lag1, degree);
39    pnew(myp, degree);
40    pnew(ptmp, degree);
41    DEGREE(lag1) = 0;
42    cset(COEFF(lag1, 0), 1, 0);
43    DEGREE(myp) = 1;
44    cset(COEFF(myp, 1), -1, 0);
45    cset(COEFF(myp, 0), 1, 0);
46    for (i = 2; i <= degree; i++) {
47      pcopy(lag0, lag1);
48      pcopy(lag1, myp);
49      DEGREE(ptmp) = 1;
50      cset(COEFF(ptmp, 1), -1, 0);
51      cset(COEFF(ptmp, 0), 2*i-1, 0);
52      pmul(ptmp, ptmp, lag1);
53      pimul(lag0, (i-1)*(i-1), lag0);
54      psub(myp, ptmp, lag0);
55    }
56    printf("# Used lbuild -p%d\n", precision);
57    pdump(myp);
58    pfree(lag0);
59    pfree(lag1);
```

```
60      pfree(ptmp);
61    }
```

C.4.5 mbuild.c

This program builds the coefficients of a Mandelbrot polynomial using the recurrence relation given in [BF00a, p. 3].

```
1   /*
2    * Build a Mandelbrot polynomial
3    */
4   #include <math.h>
5   #include <stdio.h>
6   #include <stdlib.h>
7   #include "mylib.h"
8   #include "poly.h"
9   #include "zero.h"
10
11  char *optarg;
12  int   opterr, optind, optopt;
13
14  /*
15   * Build a Mandelbrot polynomial from the recursive formula
16   * given in Bini (p. 5)
17   */
18  int main(int argc, char *argv[]) {
19    int  c;
20    int  degree = 0;
21    int  i;
22    int  k;
23    Poly ptmp;
24    double r;
25    while ((c = getopt(argc, argv, "n:p:")) != -1) {
26      switch (c) {
27      case 'n':
28        degree = atoi(optarg);
29        break;
30      case 'p':
31        precision = atoi(optarg);
32        break;
33      default:
34        fatal("Unknown option %s", argv[optind-1]);
35      }
36    }
37    rinit(precision);
38    r = log(degree+1)/log(2);
39    k = (int)r;
```

```
40      if (k != log(degree+1)/log(2)) {
41        printf("Invalid Mandelbrot degree %d\n", degree);
42        exit(EXIT_FAILURE);
43      }
44      pnew(myp, degree);
45      DEGREE(myp) = 0;
46      cset(COEFF(myp, 0), 1, 0);
47      pnew(ptmp, 1);
48      cset(COEFF(ptmp, 1), 1, 0);
49      for (i = 1; i <= k; i++) {
50        pmul(myp, myp, myp);
51        pmul(myp, ptmp, myp);
52        cset(COEFF(myp,0), 1, 0);
53      }
54      printf("# Used mbuild -p%d\n", precision);
55      pdump(myp);
56      pfree(ptmp);
57    }
```

C.4.6 validate.c

This program performs an extremely crude validation on a set of approximate zeros. It computes their sum and their product and prints these values. They should equal the coefficients a_{n-1} and a_0, respectively, of the monic polynomial, see Equation (2.4) on page 6.

```
1   /*
2    * Crudely validate a polynomial (on stdout) from
3    * its zeros (on stdin).  Check their sum and product.
4    */
5   #include "mylib.h"
6   #include "zero.h"
7
8   char *optarg;
9   int   opterr, optind, optopt;
10
11  int main(int argc, char *argv[]) {
12    int     c;
13    Complex product;
14    Complex sum;
15    int     i;
16    Zero    z;
17    while ((c = getopt(argc, argv, "p:")) != -1) {
18      switch (c) {
19      case 'p':
20        precision = atoi(optarg);
21        break;
```

```
22        default:
23          fatal("Unknown option %s", argv[optind-1]);
24        }
25      }
26      rinit(precision);
27      zinput(z);
28      printf("Zeros\n");
29      zoutput(z);
30      cnew(product, 1, 0);
31      cnew(sum, 0, 0);
32      for (i = 0; i < ZEROS(z); i++) {
33        int j;
34        for (j = 0; j < MULT(z, i); j++) {
35          cadd(sum, sum, ZERO(z, i));
36          cmul(product, product, ZERO(z, i));
37        }
38      }
39      printf("Used validate -p%d\n", precision);
40      printf("Sum     = ");
41      coutput(sum);
42      printf("\nProduct = ");
43      coutput(product);
44      printf("\n");
45      cfree(product);
46      cfree(sum);
47    }
```

C.4.7 zbuild.c

This program builds the coefficients of an arbitrary polynomial from a collection of complex zero values, including their multiplicities.

```
1   /*
2    * Build a polynomial (on stdout) from its zeros (on stdin)
3    */
4   #include "mylib.h"
5   #include "poly.h"
6   #include "zero.h"
7
8   char *optarg;
9   int   opterr, optind, optopt;
10
11  int main(int argc, char *argv[]) {
12    int     c;
13    Complex ctmp;
14    int     degree = 0;
15    int     i;
```

```
16    Zero    z;
17    while ((c = getopt(argc, argv, "p:")) != -1) {
18      switch (c) {
19      case 'p':
20        precision = atoi(optarg);
21        break;
22      default:
23        fatal("Unknown option %s", argv[optind-1]);
24      }
25    }
26    rinit(precision);
27    zinput(z);
28    printf("Zeros\n");
29    zoutput(z);
30    pnew(myp, zdegree(z));
31    cnew(COEFF(myp, 0), 1, 0);
32    cnew(ctmp, 0, 0);
33    for (i = 0; i < ZEROS(z); i++) {
34      int j;
35      for (j = 0; j < MULT(z, i); j++) {
36        int k;
37        degree++;
38        cnew(COEFF(myp, degree), 1, 0);
39        for (k = degree-1; k > 0; k--) {
40          cmul(ctmp, COEFF(myp, k), ZERO(z, i));
41          csub(COEFF(myp, k), COEFF(myp, k-1), ctmp);
42        }
43        cmul(ctmp, COEFF(myp, 0), ZERO(z, i));
44        cneg(COEFF(myp, 0), ctmp);
45      }
46    }
47    printf("# Used zbuild -p%d\n", precision);
48    pdump(myp);
49    cfree(ctmp);
50    pfree(myp);
51  }
```

C.5 Makefile

This section describes the `Makefile` for maintaining the executable files used by our algorithm. Needless to say that another `Makefile` is used for maintaining an up-to-date version of this thesis. Details of the `make` program can be found in [Fou10].

```
1  CC     = gcc
2  CFLAGS = -ansi -g
```

```
3   DEPEND = makedepend
4   EXES   = cbuild ebuild ehrlich3 farmer3 farmer4 \
5            farmer5 farmerv \
6            fbuild hansen3 kbuild lbuild mbuild rall2 \
7            sbuild search traub3 validate zbuild
8   HDRS   = complex.h ehrlich3.h farmer3.h farmer4.h \
9            farmer5.h farmerv.h \
10           hansen3.h mylib.h poly.h rall2.h \
11           real.h search.h traub3.h zero.h
12  LIBS   = -lgmp -lm -lrt
13  OBJS   = complex.o mylib.o poly.o real.o zero.o
14  SRCS   = cbuild.c complex.c ehrlich3.c farmer3.c farmer4.c \
15           farmer5.c farmerv.c hansen3.c kbuild.c mylib.c \
16           poly.c rall2.c real.c sbuild.c search.c traub3.c \
17           validate.c zero.c
18
19  # It says it all
20  #
21  all: $(EXES)
22
23  # Generic rules (GNU Make only)
24  #
25  %.c:            %.h
26          touch $@
27  %.o:            %.c
28          $(CC) $(CFLAGS) -c $<
29
30  # Mylib infrastracture
31  #
32  mylib.h:        poly.h zero.h
33          touch mylib.h
34
35  # Complex arithmetic infrastracture
36  #
37  complex.h:      real.h
38          touch complex.h
39  complex.c:      complex.h mylib.h
40          touch complex.c
41
42  # Polynomial arithmetic infrastracture
43  #
44  poly.h:         complex.h
45          touch poly.h
46
47  # Real arithmetic
48  #
49  real.c:         mylib.h real.h
50          touch real.c
51
52  # Search complex plane
```

```
53  #
54  search.h:       poly.h zero.h
55          touch search.h
56  search.c:       mylib.h search.h
57          touch search.c
58  search:         search.o $(OBJS)
59          $(CC) $(LIBS) -o $@ $^
60
61  # Zero arithmetic infrastracture
62  #
63  zero.h:         complex.h
64          touch zero.h
65
66  # Rall's second-order modified Newton method
67  #
68  rall2.c:        mylib.h complex.h rall2.h poly.h zero.h
69          touch rall2.c
70  rall2:          rall2.o $(OBJS)
71          $(CC) $(LIBS) -o $@ $^
72
73  # Ehrlich's third-order IF
74  #
75  ehrlich3.c:     mylib.h complex.h poly.h ehrlich3.h zero.h
76          touch ehrlich3.c
77  ehrlich3:       ehrlich3.o $(OBJS)
78          $(CC) $(LIBS) -o $@ $^
79
80  # Farmer's third-order IF
81  #
82  farmer3.c:      mylib.h complex.h farmer3.h poly.h zero.h
83          touch farmer3.c
84  farmer3:        farmer3.o $(OBJS)
85          $(CC) $(LIBS) -o $@ $^
86
87  # Farmer's fourth-order IF
88  #
89  farmer4.c:      mylib.h complex.h farmer4.h poly.h zero.h
90          touch farmer4.c
91  farmer4:        farmer4.o $(OBJS)
92          $(CC) $(LIBS) -o $@ $^
93
94  # Farmer's fifth-order IF
95  #
96  farmer5.c:      mylib.h complex.h farmer5.h poly.h zero.h
97          touch farmer5.c
98  farmer5:        farmer5.o $(OBJS)
99          $(CC) $(LIBS) -o $@ $^
100
101 # Farmer's variable-order IF
102 #
```

```
103 farmerv.c:      mylib.h complex.h farmerv.h poly.h zero.h
104         touch farmerv.c
105 farmerv:        farmerv.o $(OBJS)
106         $(CC) $(LIBS) -o $@ $^
107
108 # Hansen's third-order IF
109 #
110 hansen3.c:      mylib.h complex.h poly.h hansen3.h zero.h
111         touch hansen3.c
112 hansen3:        hansen3.o $(OBJS)
113         $(CC) $(LIBS) -o $@ $^
114
115 # Traub's third-order IF (from book p.139)
116 #
117 traub3.c:       mylib.h complex.h poly.h traub3.h zero.h
118         touch traub3.c
119 traub3:         traub3.o $(OBJS)
120         $(CC) $(LIBS) -o $@ $^
121
122 # Build polynomial from given coefficients
123 cbuild.c:       mylib.h poly.h
124         touch cbuild.c
125 cbuild:         cbuild.o $(OBJS)
126         $(CC) $(LIBS) -o $@ $^
127
128 # Build Eulerian polynomial
129 #
130 ebuild.c:       mylib.h poly.h
131         touch ebuild.c
132 ebuild:         ebuild.o $(OBJS)
133         $(CC) $(LIBS) -o $@ $^
134
135 # Build polynomial from the factors
136 #
137 fbuild.c:       mylib.h poly.h
138         touch fbuild.c
139 fbuild:         fbuild.o $(OBJS)
140         $(CC) $(LIBS) -o $@ $^
141
142 # Build Kirrinnis polynomial
143 #
144 kbuild.c:       mylib.h poly.h
145         touch kbuild.c
146 kbuild:         kbuild.o $(OBJS)
147         $(CC) $(LIBS) -o $@ $^
148
149 # Build Laguerre polynomial
150 #
151 lbuild.c:       mylib.h poly.h
152         touch lbuild.c
```

```
153  lbuild:         lbuild.o $(OBJS)
154          $(CC) $(LIBS) -o $@ $^
155
156  # Build Mandelbrot polynomial
157  #
158  mbuild.c:       mylib.h poly.h
159          touch mbuild.c
160  mbuild:         mbuild.o $(OBJS)
161          $(CC) $(LIBS) -o $@ $^
162
163  # Build spiral polynomial
164  #
165  sbuild.c:       mylib.h poly.h
166          touch sbuild.c
167  sbuild:         sbuild.o $(OBJS)
168          $(CC) $(LIBS) -o $@ $^
169
170  # Validate polynomial from the zeros
171  #
172  validate.c:     mylib.h zero.h
173          touch validate.c
174  validate:       validate.o $(OBJS)
175          $(CC) $(LIBS) -o $@ $^
176
177  # Build polynomial from the zeros
178  #
179  zbuild.c:       mylib.h complex.h poly.h zero.h
180          touch zbuild.c
181  zbuild:         zbuild.o $(OBJS)
182          $(CC) $(LIBS) -o $@ $^
183
184  # Clean up
185  #
186  clean:
187          rm *~ *.o
188
189  # Dependencies
190  #
191  depend: $(SRCS)
192          $(DEPEND) $(CFLAGS) $(SRCS)
193
194  # Print documents
195  #
196  print: $(HDRS) $(SRCS)
197          pr $^ | lpr
198
199  # DO NOT DELETE THIS LINE -- makedepend depends on it.
```

C.6 Matlab Program Files

This section describes the various Matlab [Mat12] programs used to generate and verify the equations and IFs used for implementing the second stage of our algorithm. They are heavy users of the symbolic manipulation module.

C.6.1 multiple.m

This program file generates symbolic equations for our multiple IFs, both one-point and simultaneous, together with their orders of convergence and asmyptotic error constants.

```
% Multiple IFs.  Polynomial format rather than rational
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
time = clock();
fprintf('Begin program %i:%i:%i\n', ...
  time(4), time(5), round(time(6)));
% Define symbols with the following notation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% u  = Taylor series expansion of u(z) about z.
% ua = Taylor series expansion of u(z) about alpha.
% Ditto for all other variables.

syms A2 A3 A4 A5;
syms B2 B3 B4 B5 B6;
syms C2 C3 C4 C5 C6 C7 C8 C9;
syms C2a C3a C4a C5a C6a C7a C8a C9a;
syms eps;
syms hat2 hat3 hat4 hat5;
syms Lambda2 Lambda3 Lambda4 Lambda5;
syms m;
syms oeps2 oeps3 oeps4 oeps5;
syms ohat2 ohat3 ohat4 ohat5;
syms p pa p1a p2a p3a;
syms seps3 seps4 seps5;
syms shat3 shat4 shat5;
syms S1 S2 S3;
syms S1a S2a S3a;
syms SUM(n);
syms T1 T2 T3;
syms T1a T2a T3a;
syms u U;
syms z;
time = clock();
fprintf('End definitions %i:%i:%i\n', ...
  time(4), time(5), round(time(6)));
```

```
35
36  % Step 1.  Build Lambda_i(z)
37  %%%%%%%%%%%%%%%%%%%%%%%%%%%%
38  % Equation (18)
39  Lambda2 = U;
40  Lambda3 = U ...
41    + B2*Lambda2^2;
42  Lambda4 = U ...
43    + B2*Lambda3^2 ...
44    - B3*Lambda2^3;
45  Lambda5 = U ...
46    + B2*Lambda4^2 ...
47    - B3*Lambda3^3 ...
48    + B4*Lambda2^4;
49  time = clock();
50  fprintf('End step 1 %i:%i:%i\n', ...
51    time(4), time(5), round(time(6)));
52
53  % Step 2.  Build basic equations
54  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
55  % Equation (20)
56  basic2 = z - collect(Lambda2, U);
57  basic3 = z - collect(Lambda3, U);
58  basic4 = z - collect(Lambda4, U);
59  basic5 = z - collect(Lambda5, U);
60  for n = 4 : 4
61    basic4 = subs(basic4, U^n, 0);
62  end
63  for n = 5 : 8
64    basic5 = subs(basic5, U^n, 0);
65  end
66  time = clock();
67  fprintf('End step 2 %i:%i:%i\n', ...
68    time(4), time(5), round(time(6)));
69
70  % Step 3.  Build one-point IFs with A_i(z) and m*u(z)
71  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
72  % Equation (21)
73  B2 = -(m-1)/(factorial(2)*m*u)+A2;
74  B3 = (m-1)*(2*m-1)/(factorial(3)*(m*u)^2) ...
75    -(m-1)/(m*u)*A2+A3;
76  B4 = -(m-1)*(2*m-1)*(3*m-1)/(factorial(4)*(m*u)^3) ...
77    +(m-1)*(2*m-1)/(factorial(2)*(m*u)^2)*A2 ...
78    -(m-1)/(m*u)*(A2^2/factorial(2)+A3)+A4;
79  B5 = (m-1)*(2*m-1)*(3*m-1)*(4*m-1)/(factorial(5)*(m*u)^4) ...
80    -(m-1)*(2*m-1)*(3*m-1)/(factorial(3)*(m*n)^3)*A2 ...
81    +(m-1)*(2*m-1)/(factorial(2)*(m*u)^2)*(A2^2+A3) ...
82    -(m-1)/(m*u)*(A2*A3+A4)+A5;
83  U = m*u;
84
```

```
% Schroder's second-order
%%%%%%%%%%%%%%%%%%%%%%%%%
% Equation (22)
ohat2 = collect(eval(basic2), u);

% Hansen's third-order
%%%%%%%%%%%%%%%%%%%%%%
% Equation (23)
ohat3 = collect(eval(basic3), u);

% Traub's fourth-order
%%%%%%%%%%%%%%%%%%%%%%
% Equation (24)
ohat4 = collect(eval(basic4), u);

% Traub's fifth-order
%%%%%%%%%%%%%%%%%%%%%%
% Equation (25)
ohat5 = collect(eval(basic5), u);
time = clock();
fprintf('End step 3 %i:%i:%i\n', ...
  time(4), time(5), round(time(6)));

% Step 4.  Build simultaneous IFs
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Derivatives of B_i(z)
%%%%%%%%%%%%%%%%%%%%%%%
B21 = 3*B3 - 2*B2^2;
B31 = 4*B4 - 2*B2*B3;
B41 = 5*B5 - 2*B2*B4;
%B51 = 6*B6 - 2*B2*B5;
B22 = 3*B31 - 4*B2*B21;
B32 = 4*B41 - 2*B21*B3 - 2*B2*B31;
%B42 = 5*B51 - 2*B21*B4 - 2*B2*B41;
B23 = 3*B32 - 4*B21^2 - 4*B2*B22;

% S_i(z)
%%%%%%%%
S10 = m*(B2 ...
  + (B2^2 - B3)*eps ...
  + (B2^3 - 2*B2*B3 + B4)*eps^2 ...
  + (B2^4 - 3*B2^2*B3 + B2*B4 ...
    + B3^2 - B5)*eps^3);
S11 = m*(B21 ...
  + B2^2 - B3 + (2*B2*B21 - B31)*eps ...
  + 2*(B2^3 - 2*B2*B3 + B4)*eps ...
  + 3*(B2^4 - 3*B2^2*B3 + B2*B4 ...
    + B3^2 - B5)*eps^2);
S12 = m*(B22 ...
```

```
    + 2*B2*B21 - B31 + 2*B2*B21 - B31 ...
    + 2*(B2^3 - 2*B2*B3 + B4) ...
    + 6*(B2^4 - 3*B2^2*B3 + B2*B4 ...
      + B3^2 - B5)*eps);
  S13 = m*(B23 ...
    + 2*B21^2 + 2*B2*B22 - B32 ...
      + 2*B21^2 + 2*B2*B22 - B32 ...
    + 2*(3*B2^2*B21 - 2*B21*B3 - 2*B2*B31 ...
       + B41) ...
    + 6*(B2^4 - 3*B2^2*B3 + B2*B4 ...
       + B3^2 - B5));
  S1 = subs(S10, eps, 0);
  S2 = subs(-S11, eps, 0);
  S3 = subs(S12/2, eps, 0);
  S4 = -S13/3;


  % Farmer & Loizou's third-order
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % Equation (36)
  B2 = T1/m;
  shat3 = collect(eval(basic3), u);
  B2 = -(m-1)/(factorial(2)*m*u)+A2;

  % Farmer & Loizou's fourth-order
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % Equation (37)
  B3 = (B2^2 - T2/m)/2;
  shat4 = collect(eval(basic4), u);
  B3 = (m-1)*(2*m-1)/(factorial(3)*(m*u)^2) ...
    -(m-1)/(m*u)*A2+A3;

  % Farmer & Loizou's fifth-order
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % Equation (38)
  B4 = (-B2^3+3*B2*B3+T3/m)/3;
  shat5 = collect(eval(basic5), u);
  B4 = -(m-1)*(2*m-1)*(3*m-1)/(factorial(4)*(m*u)^3) ...
    +(m-1)*(2*m-1)/(factorial(2)*(m*u)^2)*A2 ...
    -(m-1)/(m*u)*(A2^2/factorial(2)+A3)+A4;

  time = clock();
  fprintf('End step 4 %i:%i:%i\n', ...
    time(4), time(5), round(time(6)));

  % Rational third-order
  %%%%%%%%%%%%%%%%%%%%%%
  rhat3 = z - m*u ...
    - (m*u^2)/(1 - T1*u);

```

```
% Step 5.  Orders of convergence
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Equation (47)
% Note!  All base/primitive equations divided by
% p^(m)(alpha)eps^{m-1}/m!  It makes life easier.

% Define p(z)@alpha and its derivatives
% Equation (47)
pa = eps/m ...
  + C2a*eps^2 ...
  + C3a*eps^3 ...
  + C4a*eps^4 ...
  + C5a*eps^5;
time = clock();
fprintf('End p(z)@alpha %i:%i:%i\n', ...
  time(4), time(5), round(time(6)));

p1a = 1 ...
  + (m+1)*C2a*eps ...
  + (m+2)*C3a*eps^2 ...
  + (m+3)*C4a*eps^3 ...
  + (m+4)*C5a*eps^4 ...
  + (m+5)*C6a*eps^5;
time = clock();
fprintf('End p''(z)@alpha %i:%i:%i\n', ...
  time(4), time(5), round(time(6)));

p2a = (m-1)/eps ...
  + m*(m+1)*C2a ...
  + (m+1)*(m+2)*C3a*eps ...
  + (m+2)*(m+3)*C4a*eps^2 ...
  + (m+3)*(m+4)*C5a*eps^3 ...
  + (m+4)*(m+5)*C6a*eps^4 ...
  + (m+5)*(m+6)*C7a*eps^5;
time = clock();
fprintf('End p''''(z)@alpha %i:%i:%i\n', ...
  time(4), time(5), round(time(6)));

p3a = (m-2)*(m-1)/eps^2 ...
  + (m-1)*m*(m+1)*C2a/eps ...
  + m*(m+1)*(m+2)*C3a ...
  + (m+1)*(m+2)*(m+3)*C4a*eps ...
  + (m+2)*(m+3)*(m+4)*C5a*eps^2 ...
  + (m+3)*(m+4)*(m+5)*C6a*eps^3 ...
  + (m+4)*(m+5)*(m+6)*C7a*eps^4 ...
  + (m+5)*(m+6)*(m+7)*C8a*eps^5;
time = clock();
fprintf('End p''''''(z)@alpha %i:%i:%i\n', ...
  time(4), time(5), round(time(6)));

```

```
p4a = (m-3)*(m-2)*(m-1)/eps^3 ...
  + (m-2)*(m-1)*m*(m+1)*C2a/eps^2 ...
  + (m-1)*m*(m+1)*(m+2)*C3a/eps ...
  + m*(m+1)*(m+2)*(m+3)*C4a ...
  + (m+1)*(m+2)*(m+3)*(m+4)*C5a*eps ...
  + (m+2)*(m+3)*(m+4)*(m+5)*C6a*eps^2 ...
  + (m+3)*(m+4)*(m+5)*(m+6)*C7a*eps^3 ...
  + (m+4)*(m+5)*(m+6)*(m+7)*C8a*eps^4 ...
  + (m+5)*(m+6)*(m+7)*(m+8)*C9a*eps^5;
time = clock();
fprintf('End p''''''''(z)@alpha %i:%i:%i\n', ...
  time(4), time(5), round(time(6)));

recip = collect(taylor(1/p1a, eps), eps);

% Define u(z)@alpha
% Equation (48)
ua = simplify(pa*recip);
ua = collect(ua, eps);
for n = 6 : 10
  ua = subs(ua, eps^n, 0);
end
time = clock();
fprintf('End u(z)@alpha %i:%i:%i\n', ...
  time(4), time(5), round(time(6)));

% Define A_k(z)@alpha
% Equation (50)
A2a = eps*p2a*recip/factorial(2);
A2a = collect(A2a, eps);
for n = 5 : 11
  A2a = subs(A2a, eps^n, 0);
end
A2a = A2a/eps;
time = clock();
fprintf('End A2(z)@alpha %i:%i:%i\n', ...
  time(4), time(5), round(time(6)));

A3a = eps^2*p3a*recip/factorial(3);
A3a = collect(A3a, eps);
for n = 5 : 12
  A3a = subs(A3a, eps^n, 0);
end
A3a = A3a/eps^2;
time = clock();
fprintf('End A3(z)@alpha %i:%i:%i\n', ...
  time(4), time(5), round(time(6)));

A4a = eps^3*p4a*recip/factorial(4);
A4a = collect(A4a, eps);
```

```
285  for n = 5 : 13
286    A4a = subs(A4a, eps^n, 0);
287  end
288  A4a = A4a/eps^3;
289  time = clock();
290  fprintf('End A4(z)@alpha %i:%i:%i\n', ...
291    time(4), time(5), round(time(6)));
292
293  % Define S_k(z)@alpha
294  %%%%%%%%%%%%%%%%%%
295  % Equation (52)
296  S1a = m*C2a;
297  time = clock();
298  fprintf('End S1(z)@alpha %i:%i:%i\n', ...
299    time(4), time(5), round(time(6)));
300
301  S2a = m*(m*C2a^2 ...
302    - 2*C3a);
303  time = clock();
304  fprintf('End S2(z)@alpha %i:%i:%i\n', ...
305    time(4), time(5), round(time(6)));
306
307  S3a = m*(m^2*C2a^3 ...
308    - 3*m*C2a*C3a ...
309    + 3*C4a);
310  time = clock();
311  fprintf('End S3(z)@alpha %i:%i:%i\n', ...
312    time(4), time(5), round(time(6)));
313
314  S4a = m*(m^3*C2a^4 ...
315    - 4*m^2*C2a^2*C3a ...
316    + 4*m*C2a*C4a ...
317    + 2*m*C3a^2 ...
318    - 4*C5a);
319  time = clock();
320  fprintf('End S4(z)@alpha %i:%i:%i\n', ...
321    time(4), time(5), round(time(6)));
322
323  % Define T_k(z)@alpha
324  T1a = simplify(S1a ...
325    - S2a*eps ...
326    + SUM(2));
327  T1a = collect(T1a, eps);
328  for n = 6 : 10
329    T1a = subs(T1a, eps^n, 0);
330  end
331  time = clock();
332  fprintf('End T1(z)@alpha %i:%i:%i\n', ...
333    time(4), time(5), round(time(6)));
334
```

```
T2a = simplify(S2a ...
  - 2*S3a*eps ...
  + 2*SUM(3));
T2a = collect(T2a, eps);
for n = 6 : 10
  T2a = subs(T2a, eps^n, 0);
end
time = clock();
fprintf('End T2(z)@alpha %i:%i:%i\n', ...
  time(4), time(5), round(time(6)));

T3a = simplify(S3a ...
  - 3*S4a*eps ...
  + 3*SUM(4));
T3a = collect(T3a, eps);
for n = 6 : 10
  T3a = subs(T3a, eps^n, 0);
end
time = clock();
fprintf('End T3(z)@alpha %i:%i:%i\n', ...
  time(4), time(5), round(time(6)));
time = clock();
fprintf('End step 5 %i:%i:%i\n', ...
  time(4), time(5), round(time(6)));

% Step 6.  One-point IFs
%%%%%%%%%%%%%%%%%%%%%%%%

A2 = A2a;
A3 = A3a;
A4 = A4a;
u = ua;
z = eps;

% Schroder's second-order
%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Equation (54)
oeps2 = ohat2;
oeps2 = collect(eval(oeps2), eps);
for n = 3 : 5
  oeps2 = subs(oeps2, eps^n, 0);
end
oeps2 = simplify(oeps2);
time = clock();
fprintf('End Rall(2) %i:%i:%i\n', ...
  time(4), time(5), round(time(6)));

% Ehrlich's third-order
%%%%%%%%%%%%%%%%%%%%%%%
% Equation (55)
```

```
385   oeps3 = ohat3;
386   oeps3 = collect(eval(oeps3), eps);
387   for n = 4 : 13
388     oeps3 = subs(oeps3, eps^n, 0);
389   end
390   oeps3 = simplify(oeps3);
391   time = clock();
392   fprintf('End Ehrlich(3) %i:%i:%i\n', ...
393     time(4), time(5), round(time(6)));
394
395   % Traub's fourth-order
396   %%%%%%%%%%%%%%%%%%%%%%
397   % Equation (56)
398   oeps4 = ohat4;
399   oeps4 = collect(eval(oeps4), eps);
400   for n = 5 : 21
401     oeps4 = subs(oeps4, eps^n, 0);
402   end
403   oeps4 = simplify(oeps4);
404   time = clock();
405   fprintf('End Traub(4) %i:%i:%i\n', ...
406     time(4), time(5), round(time(6)));
407
408   % Traub's fifth-order
409   %%%%%%%%%%%%%%%%%%%%%%
410   % Equation (57)
411   oeps5 = ohat5;
412   oeps5 = collect(eval(oeps5), eps);
413   for n = 6 : 29
414     oeps5 = subs(oeps5, eps^n, 0);
415   end
416   oeps5 = simplify(oeps5);
417   time = clock();
418   fprintf('End Traub(5) %i:%i:%i\n', ...
419     time(4), time(5), round(time(6)));
420   time = clock();
421   fprintf('End step 6 %i:%i:%i\n', ...
422     time(4), time(5), round(time(6)));
423
424   % Step 7.  Simultaneous IFs
425   %%%%%%%%%%%%%%%%%%%%%%%%%%
426
427   T1 = T1a;
428   T2 = T2a;
429   T3 = T3a;
430
431   % No second-order in polynomial form
432
433   % Farmer & Loizou's third-order
434   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Equation (58)
seps3 = shat3;
seps3 = collect(eval(seps3), eps);
for n = 4 : 11
  seps3 = subs(seps3, eps^n, 0);
end
seps3 = simplify(seps3);
time = clock();
fprintf('End Farmer(3) %i:%i:%i\n', ...
  time(4), time(5), round(time(6)));

% Farmer & Loizou's fourth-order
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Equation (59)
seps4 = shat4;
seps4 = collect(eval(seps4), eps);
for n = 5 : 21
  seps4 = subs(seps4, eps^n, 0);
end
seps4 = simplify(seps4);
time = clock();
fprintf('End Farmer(4) %i:%i:%i\n', ...
  time(4), time(5), round(time(6)));

% Farmer & Loizou's fifth-order
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Equation (60)
seps5 = shat5;
seps5 = collect(eval(seps5), eps);
for n = 6 : 29
  seps5 = subs(seps5, eps^n, 0);
end
seps5 = simplify(seps5);
time = clock();
fprintf('End Farmer(5) %i:%i:%i\n', ...
  time(4), time(5), round(time(6)));

% Rational third-order
%%%%%%%%%%%%%%%%%%%%%%
numer = m*u;
denom = 1 - T1*u;
recip = taylor(1/denom, eps);
reps3 = numer*recip;
for n = 4 : 11
  reps3 = subs(reps3, eps^n, 0);
end
reps3 = collect(reps3, eps);
for n = 4 : 6
  reps3 = subs(reps3, eps^n, 0);
end
```

```
485  reps3 = simplify(reps3);
486
487  time = clock();
488  fprintf('End step 7 %i:%i:%i\n', ...
489    time(4), time(5), round(time(6)));
490
```

C.6.2 simple.m

This program file generates symbolic equations for our simple IFs, both one-point and simultaneous, together with their orders of convergence.

This listing has only been included for completeness as all IFs, together with their orders of convergence and asymptotic error constants, can be obtained from the Matlab program for multiple zeros, see §C.6.1 on page 190.

```
1   % Simple IFs.  Polynomial form rather than rational
2   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
3
4   % Define basic symbols
5   syms alpha eps z;
6
7   % Define basic functions
8   syms A2(z) A3(z) A4(z) A5(z) A6(z) A7(z);
9   syms S1(z) S2(z) S3(z) S4(z) S5(z);
10  syms T1(z) T2(z) T3(z);
11  syms p(z) SUM(n) u(z);
12
13  % Step 1.  Build lambda_i(z)
14  %%%%%%%%%%%%%%%%%%%%%%%%%%%%
15  lambda2(z) = u(z);
16  lambda3(z) = u(z) ...
17    + A2(z)*lambda2(z)^2;
18  lambda4(z) = u(z) ...
19    + A2(z)*lambda3(z)^2 ...
20    - A3(z)*lambda2(z)^3;
21  lambda5(z) = u(z) ...
22    + A2(z)*lambda4(z)^2 ...
23    - A3(z)*lambda3(z)^3 ...
24    + A4(z)*lambda2(z)^4;
25  lambda6(z) = u(z) ...
26    + A2(z)*lambda5(z)^2 ...
27    - A3(z)*lambda4(z)^3 ...
28    + A4(z)*lambda3(z)^4 ...
29    - A5(z)*lambda2(z)^5;
30
31  % Step 2.  Build one-point IFs
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Newton's second-order
% Equation (5.13)
ohat2(z) = z - lambda2(z);

% Halley's third-order
% Equation (5.14)
ohat3(z) = z - lambda3(z);

% Kiss' fourth-order
% Equation (5.15)
ohat4(z) = z - collect(lambda4(z), u(z));
for n = 4 : 4
  ohat4(z) = subs(ohat4(z), u(z)^n, 0);
end

% Kiss' fifth-order
% Equation (5.16)
ohat5(z) = z - collect(lambda5(z), u(z));
for n = 5 : 8
  ohat5(z) = subs(ohat5(z), u(z)^n, 0);
end

% Step 3.  Build simultaneous IFs
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% No second-order in polynomial form

% Farmer & Loizou's third-order
% Equation (5.24)
shat3(z) = subs(ohat3(z), A2(z), T1(z));
shat3(z) = simplify(shat3(z));

% Farmer & Loizou's fourth-order
% Equation (5.25)
shat4(z) = subs(ohat4(z), A3(z), (A2(z)^2 ...
  - T2(z))/2);
shat4(z) = simplify(shat4(z));

% Farmer & Loizou's fifth-order
% Equation (5.26)
shat5(z) = subs(ohat5(z), A4(z), ...
  (-A2(z)^3 + 3*A2(z)*A3(z) + T3(z))/3);
shat5(z) = simplify(shat5(z));

% Orders of Convergence
%%%%%%%%%%%%%%%%%%%%%%%%%
% Also useful for verification of equations

```

```
% Define first derivatives of Ai(z) (second digit)
% Equation (A.4)
A21(z) = 3*A3(z) - 2*A2(z)^2;
A31(z) = 4*A4(z) - 2*A2(z)*A3(z);
A41(z) = 5*A5(z) - 2*A2(z)*A4(z);
A51(z) = 6*A6(z) - 2*A2(z)*A5(z);
A61(z) = 7*A7(z) - 2*A2(z)*A6(z);

% Define second derivatives of Ai(z) (second digit)
% Equation (A.5)
A22(z) = 3*A31(z) - 4*A2(z)*A21(z);
A32(z) = 4*A41(z) - 2*A21(z)*A3(z) ...
  - 2*A2(z)*A31(z);
A42(z) = 5*A51(z) - 2*A21(z)*A4(z) ...
  - 2*A2(z)*A41(z);
A52(z) = 6*A61(z) - 2*A21(z)*A5(z) ...
  - 2*A2(z)*A51(z);

% Define third derivatives of Ai(z) (second digit)
% Equation (A.6)
A23(z) = 3*A32(z) - 4*A21(z)^2 ...
  - 4*A2(z)*A22(z);
A33(z) = 4*A42(z) - 2*A22(z)*A3(z) ...
  - 4*A21(z)*A31(z) - 2*A2(z)*A32(z);
A43(z) = 5*A52(z) - 2*A22(z)*A4(z)...
  - 4*A21(z)*A41(z) - 2*A2(z)*A42(z);

% Define fourth derivatives of Ai(z) (second digit)
% Equation (A.7)
A24(z) = 3*A33(z) - 12*A21(z)*A22(z) ...
  - 4*A2(z)*A23(z);
A34(z) = 4*A43(z) - 2*A23(z)*A3(z) ...
  - 6*A22(z)*A31(z) - 6*A21(z)*A32(z) ...
  - 2*A2(z)*A33(z);

% Define derivatives of Ai(z)@alpha
% Equation (A.8)
A2a = A2(alpha) + A21(alpha)*eps ...
  + A22(alpha)*eps^2/factorial(2) ...
  + A23(alpha)*eps^3/factorial(3) ...
  + A24(alpha)*eps^4/factorial(4);
A2a = simplify(A2a);
A2a = collect(A2a, eps);
for n = 4 : 4
  A2a = subs(A2a, eps^n, 0);
end
A3a = A3(alpha) + A31(alpha)*eps ...
  + A32(alpha)*eps^2/factorial(2) ...
  + A33(alpha)*eps^3/factorial(3) ...
  + A34(alpha)*eps^4/factorial(4);
```

```
A3a = simplify(A3a);
A3a = collect(A3a, eps);
for n = 3 : 4
  A3a = subs(A3a, eps^n, 0);
end
A4a = A4(alpha) + A41(alpha)*eps;
A4a = simplify(A4a);
A4a = collect(A4a, eps);
for n = 2 : 4
  A4a = subs(A4a, eps^n, 0);
end
A5a = A5(alpha);

% Define powers of Ai(z)@alpha (second digit)
A2a2 = collect(A2a^2, eps);
for n = 6 : 10
  A2a2 = subs(A2a2, eps^n, 0);
end
A2a2 = simplify(A2a2);
A2a2 = collect(A2a2, eps);

A2a3 = collect(A2a^3, eps);
for n = 6 : 15
  A2a3 = subs(A2a3, eps^n, 0);
end
A2a3 = simplify(A2a3);
A2a3 = collect(A2a3, eps);

A2a4 = collect(A2a^4, eps);
for n = 6 : 20
  A2a4 = subs(A2a4, eps^n, 0);
end
A2a4 = simplify(A2a4);
A2a4 = collect(A2a4, eps);

A3a2 = collect(A3a^2, eps);
for n = 6 : 10
  A3a2 = subs(A3a2, eps^n, 0);
end
A3a2 = simplify(A3a2);
A3a2 = collect(A3a2, eps);

% Define derivatives of u(z)
% Equation (A.11)
u1(z) = 1 - 2*A2(z)*u(z);
u2(z) = -2*(A21(z)*u(z) ...
          + A2(z)*u1(z));
u2(z) = collect(u2(z), u(z));
u3(z) = -2*(A22(z)*u(z) + 2*A21(z)*u1(z) ...
          + A2(z)*u2(z));
```

```
u3(z) = simplify(u3(z));
u3(z) = collect(u3(z), u(z));
u4(z) = -2*(A23(z)*u(z) + 3*A22(z)*u1(z) ...
          + 3*A21(z)*u2(z) + A2(z)*u3(z));
u4(z) = simplify(u4(z));
u4(z) = collect(u4(z), u(z));
u5(z) = -2*(A24(z)*u(z) + 4*A23(z)*u1(z) ...
          + 6*A22(z)*u2(z) + 4*A21(z)*u3(z) ...
          + A2(z)*u4(z));
u5(z) = simplify(u5(z));
u5(z) = collect(u5(z), u(z));

% Define derivatives of u(z)@alpha
% Equation (A.12)
u1a = subs(u1(alpha), u(alpha), 0);
u2a = subs(u2(alpha), u(alpha), 0);
u3a = subs(u3(alpha), u(alpha), 0);
u4a = subs(u4(alpha), u(alpha), 0);
u4a = simplify(u4a);
u5a = subs(u5(alpha), u(alpha), 0);
u5a = simplify(u5a);

% Define u(z)@alpha
% Equation (A.13)
ua = u1a*eps ...
  + u2a*eps^2/factorial(2) ...
  + u3a*eps^3/factorial(3) ...
  + u4a*eps^4/factorial(4) ...
  + u5a*eps^5/factorial(5);
ua = simplify(ua);
ua = collect(ua, eps);

% Powers of ua
ua2 = simplify(ua^2);
ua2 = collect(ua2, eps);
for n = 6 : 25
  ua2 = subs(ua2, eps^n, 0);
end
ua2 = simplify(ua2);
ua2 = collect(ua2, eps);
ua3 = simplify(ua^3);
ua3 = collect(ua3, eps);
for n = 6 : 125
  ua3 = subs(ua3, eps^n, 0);
end
ua3 = simplify(ua3);
ua3 = collect(ua3, eps);
ua4 = simplify(ua^4);
ua4 = collect(ua4, eps);
for n = 6 : 625
```

```
232     ua4 = subs(ua4, eps^n, 0);
233   end
234   ua4 = simplify(ua4);
235   ua4 = collect(ua4, eps);
236   ua5 = simplify(ua^5);
237   ua5 = collect(ua5, eps);
238   for n = 6 : 3125
239     ua5 = subs(ua5, eps^n, 0);
240   end
241   ua5 = simplify(ua5);
242   ua5 = collect(ua5, eps);
243
244   % Define Tk(z)@alpha
245   T1a = A2a - (A2a2 - 2*A3a)*eps + SUM(2);
246   T1a = simplify(T1a);
247   T1a = collect(T1a, eps);
248   for n = 2 : 10
249     T1a = subs(T1a, eps^n, 0);
250   end
251   T2a = A2a2 - 2*A3a - 2*(A2a3 - 3*A2a*A3a + 3*A4a)*eps ...
252     + 2*SUM(3);
253   T2a = simplify(T2a);
254   T2a = collect(T2a, eps);
255   for n = 2 : 10
256     T2a = subs(T2a, eps^n, 0);
257   end
258   T2a = simplify(T2a);
259   T2a = collect(T2a, eps);
260   T3a = A2a3 - 3*A2a*A3a + 3*A4a ...
261     -3*(A2a4 - 4*A2a3*A3a + 4*A2a*A4a ...
262     + 2*A3a2 - 4*A5a)*eps + 3*SUM(4);
263   T3a = simplify(T3a);
264   T3a = collect(T3a, eps);
265   for n = 2 : 15
266     T3a = subs(T3a, eps^n, 0);
267   end
268   T3a = simplify(T3a);
269   T3a = collect(T3a, eps);
270
271   % Simple one-point IFs
272   %%%%%%%%%%%%%%%%%%%%%%
273   % Isaac Newton's second-order IF
274   % Equation (A.14)
275   oeps2 = simplify(eps - ua);
276   oeps2 =collect(oeps2, eps);
277   for n = 3 : 10
278     oeps2 = subs(oeps2, eps^n, 0);
279   end
280
281   % Edmond Halley's third-order IF
```

```
% Equation (A.15)
oeps3 = simplify(eps - ua - A2a*ua2);
oeps3 = collect(oeps3, eps);
for n = 4 : 15
  oeps3 = subs(oeps3, eps^n, 0);
end

% I Kiss' fourth-order IF
% Equation (A.16)
oeps4 = simplify(eps - ua - A2a*ua2 ...
  - (2*A2a2 - A3a)*ua3);
oeps4 = collect(oeps4, eps);
for n = 5 : 21
  oeps4 = subs(oeps4, eps^n, 0);
end
oeps4 = simplify(oeps4);
oeps4 = collect(oeps4, eps);

% I Kiss' fifth-order-IF
% Equation (A.17)
oeps5 = simplify(eps - ua - A2a*ua2 ...
  - (2*A2a2 - A3a)*ua3 ...
  - (5*A2a3 - 5*A2a*A3a + A4a)*ua4);
oeps5 = collect(oeps5, eps);
for n = 6 : 50
  oeps5 = subs(oeps5, eps^n, 0);
end
oeps5 = simplify(oeps5);
oeps5 = collect(oeps5, eps);

% Simple simultaneous IFs
%%%%%%%%%%%%%%%%%%%%%%%%%
% Mick Farmer and George Loizou's third-order IF
% Equation (A.22)
seps3 = simplify(eps - ua - T1a*ua2);
seps3 = collect(seps3, eps);
for n = 4 : 25
  seps3 = subs(seps3, eps^n, 0);
end
seps3 = simplify(seps3);

% Mick Farmer and George Loizou's fourth-order IF
% Equation (A.23)
seps4 = simplify(eps - ua - A2a*ua2 ...
  - (3*A2a2 + T2a)*ua3/2);
seps4 = collect(seps4, eps);
for n = 5 : 25
  seps4 = subs(seps4, eps^n, 0);
end
seps4 = simplify(seps4);
```

```
332
333  % Mick Farmer and George Loizou's fifth-order IF
334  % Equation (A.24)
335  seps5 = simplify(eps - ua - A2a*ua2 ...
336    - (2*A2a2 - A3a)*ua3 ...
337    - (14*A2a3 - 12*A2a*A3a + T3a)*ua4/3);
338  seps5 = collect(seps5, eps);
339  for n = 6 : 125
340    seps5 = subs(seps5, eps^n, 0);
341  end
342  seps5 = simplify(seps5);
343  seps5 = collect(seps5, eps);
```

C.7 Shell Program Files

This section describes the shell programs that we invoke from the command line in order to process our test polynomials and collate our results into this thesis.

C.7.1 iters.sh

This program builds the table summary of results, see §7.1 starting on page 87.

```
1   #
2   # Summarise how many iterations were taken by each IF
3   #
4   trap "exit;" 2
5   source polys.sh
6
7   inputs="../data/inputs"
8   list="$failure"
9   methods="rall2 ehrlich3 farmer3 hansen3 traub3 farmer4"
10  methods="$methods farmer5 farmerv"
11  outputs="../data/outputs"
12
13  #
14  # Process global options
15  #
16  # -B use both success and failure lists
17  # -l list of polynomials (default failure)
18  # -S use success list
19  #
20  while getopts "Bl:S" option
21  do
22    case $option in
23        B)  list=‘echo $failure $success | sed 's/ /\n/g' | sort‘
```

```
24              ;;
25          l)  list="$OPTARG"
26              ;;
27          S)  list="$success"
28              ;;
29          *)  exit 1
30              ;;
31      esac
32  done
33  shift $(($OPTIND - 1))
34  for item in $list
35  do
36    #
37    # Process all polynomials of the same degree
38    #
39    for poly in $inputs/polys/$item
40    do
41      echo -n `basename $poly`
42      input="$outputs/search/"`basename $poly`
43      output=`tail -1 $input`
44      if [ "$output" = "SEARCH FAILURE" ]
45      then
46        echo $output
47        echo " & F \\\\"
48        continue
49      fi
50      output=`echo $output | sed 's/SEARCH SUCCESS //'`
51      echo -n " & S"
52      for method in $methods
53      do
54        input="$outputs/$method/`basename $poly`"
55        output=`tail -1 $input`
56        if [ "$output" = "ITERATE FAILURE" ]
57        then
58          echo -n " & F"
59        else
60          output=`echo $output | sed 's/ITERATE SUCCESS //'`
61          echo -n " & $output"
62        fi
63        if [ -f "./${method}s" ]
64        then
65          input="$outputs/${method}s/`basename $poly`"
66          output=`tail -1 $input`
67          if [ "$output" = "ITERATE FAILURE" ]
68          then
69            echo -n " & F"
70          else
71            output=`echo $output | sed 's/ITERATE SUCCESS //'`
72            echo -n " & $output"
73          fi
```

```
74          fi
75        done
76        echo " \\\\"
77      done
78    done
```

C.7.2 polys.sh

This program divides the names of all the polynomials in our database into two shell variables, `$failure` and `$success` that are accessed by two other shell programs `iters.sh`, see §C.7.1 starting on page 207, and `solve.sh`, see §C.7.3 starting on page 209. It keeps the dichotomy between success and failure in one place.

```
1   #
2   # The test polynomials broken down into failures and
3   # successes.  For use by other scripts to stay in sync
4   #
5   failure="020o 040a"
6   success="002[a-d] 003[a-r] 004[a-s] 005[a-u] \
7     006[a-q] 007[a-l] 008[a-j] 009[a-h] 010[a-m] \
8     012[a-d] 013[a-d] 014[a-c] 015[a-h] 016[a-g] \
9     017[ab] 018[a-d] 019[ab] 020[a-n] 024[ab] \
10    028[ab] 029a 030[a-e] 031a 035a 036[ab] 038a \
11    040b 044[ab] 048a 050[a-e] 052a 055[ab] 056a 057a \
12    060[ab] 080a 100[a-e] 101a 150[ab] 164a 200a 400a"
```

C.7.3 solve.sh

This program drives our complete algorithm. Initially it runs the search stage, collecting the appropriate results. If this stage is successful, i.e. it produces a realistic set of approximations, it runs the iterative stage for each of the IFs available. Again, the results are saved and summarised.

```
1   # Search for the zeros of a polynomial (mick, 27 Jan 2008).
2   # This shell script can't be run in parallel because it
3   # uses fixed-name temporary files (/tmp/arg1, /tmp/zeros,
4   # etc.).
5   #
6   source polys.sh
7   arg1=""
8   arg2=""
9   inputs="../data/inputs"
```

```
10  list="$failure"
11  #methods="rall2 ehrlich3 farmer3 hansen3 traub3 farmer4 farmerv"
12  methods="farmer5"
13  options="yes"
14  outputs="../data/outputs"
15  search="yes"
16  # Process global options for all polynomials
17  #
18  # -c multiplier when covering smaller squares (default 3)
19  # -d debug search stage
20  #    1 = Polynomial evaluation and u(z) (mylib.c)
21  #    2 = Newton evaluation (mylib.c)
22  #    4 = Complex division (complex.c)
23  # -D debug iteration stage
24  #    1 = Polynomial evaluation and u(z) (mylib.c)
25  #    2 = Newton evaluation (mylib.c)
26  #    4 = Complex division (complex.c)
27  #    8 = Correction values (all IFs)
28  # -i iterate only, use search stage results
29  # -l list of polynomials (default $failure).
30  #    An explicit list of more than one should be quoted
31  # -m maximum outer iterations during search stage (default 8)
32  # -n maximum inner iterations during search stage (default 4)
33  # -o turn off options within polynomials
34  # -p precision during search stage (in bits)
35  # -P precision during iteration stage (in bits)
36  # -s sides during search stage (default 2 for 4 squares)
37  # -S use success list
38  # -z zeros during search stage (default 5*DEGREE(myp))
39  #
40  while getopts "c:d:D:il:m:n:op:P:s:Sz:" option
41  do
42    case $option in
43        c)  arg1="$arg1 -c$OPTARG"
44            ;;
45        d)  arg1="$arg1 -d$OPTARG"
46            ;;
47        D)  arg2="$arg2 -d$OPTARG"
48            ;;
49        i)  search="no"
50            ;;
51        l)  list="$OPTARG"
52            ;;
53        m)  arg1="$arg1 -m$OPTARG"
54            ;;
55        n)  arg1="$arg1 -n$OPTARG"
56            ;;
57        o)  options="no"
58            ;;
59        p)  arg1="$arg1 -p$OPTARG"
```

```
60              ;;
61          P)  arg2="$arg2 -p$OPTARG"
62              ;;
63          s)  arg1="$arg1 -s$OPTARG"
64              ;;
65          S)  list="$success"
66              ;;
67          z)  arg1="$arg1 -z$OPTARG"
68              ;;
69          *)  exit 1
70              ;;
71      esac
72  done
73  shift $(($OPTIND - 1))
74  echo "Arg1 $arg1"
75  echo "Arg2 $arg2"
76  # Process all polynomials in the list
77  #
78  echo "Solve $list"
79  for item in $list
80  do
81    echo "Polys $item"
82    # Process all polynomials of the same degree
83    #
84    for poly in $inputs/polys/$item
85    do
86      echo $poly
87      # Process local options for each polynomial
88      #
89      if [ $options == "yes" ]
90      then
91        opt1=""
92        if grep -m1 '# Opt1 ' $poly > /tmp/opt1
93        then
94          opt1=`cut -b7- < /tmp/opt1`
95          echo "Opt1 $opt1"
96        fi
97        opt2=""
98        if grep -m1 '# Opt2 ' $poly > /tmp/opt2
99        then
100         opt2=`cut -b7- < /tmp/opt2`
101         echo "Opt2 $opt2"
102       fi
103     fi
104     if [ $search == "yes" ]
105     then
106       # Search stage for squares containing zeros
107       #
108       output="$outputs/search/"`basename $poly`
109       ./search $opt1 $arg1 < $poly 2>&1 | tee $output
```

```
110         if tail -n1 $output | grep -v "success"
111         then
112           echo "SEARCH FAILURE" >> $output
113           continue
114         fi
115         # Save approximations
116         #
117         echo -n "SEARCH SUCCESS " >> $output
118         cat /tmp/iters >> $output
119         cp /tmp/zeros $inputs/search/`basename $poly`
120       else
121         echo "No search"
122       fi
123       # Iteration stage to improve approximations
124       #
125       echo "Opt2 $opt2"
126       echo "Arg2 $arg2"
127       for method in $methods
128       do
129         cat $poly $inputs/search/`basename $poly` > /tmp/inputs
130         output="$outputs/$method/`basename $poly`"
131         ./$method $opt2 $arg2 < /tmp/inputs > $output 2>&1
132         status=$?
133         echo -e "$method\t\t$status"
134         if [ $status -ne 0 ]
135         then
136           echo "ITERATE FAILURE" >> $output
137         else
138           echo -n "ITERATE SUCCESS " >> $output
139           cat /tmp/iters >> $output
140         fi
141         if [ -f "./${method}s" ]
142         then
143           # Option for IFs only
144           #
145           # -s use serial form of the IF
146           #
147           output="$outputs/${method}s/`basename $poly`"
148           ./$method -s $opt2 $arg2 < /tmp/inputs > $output 2>&1
149           status=$?
150           echo -e "$method (s)\t$status"
151           if [ $status -ne 0 ]
152           then
153             echo "ITERATE FAILURE" >> $output
154           else
155             echo -n "ITERATE SUCCESS " >> $output
156             cat /tmp/iters >> $output
157           fi
158         fi
159       done
```

```
160       done
161     done
```

REMARK 15. *When this is number one, we are done.*

Bibliography

[Abe73] Oliver Aberth. Iteration methods for finding all zeros of a polynomial simultaneously. *Mathematics of Computation*, 27(122):339–344, April 1973.

[AH74] Götz Alefeld and Jürgen Herzberger. On the convergence speed of some algorithms for the simultaneous approximation of polynomial zeros. *SIAM Journal of Numerical Analysis*, 11:237–243, 1974.

[Ano77] Abdel Anourein. An improvement on two iteration methods for simultaneous determination of the zeros of a polynomial. *International Journal of Computer Mathematics*, 6(3):241–252, January 1977.

[AS70] Milton Abramowitz and Irene Stegun, editors. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover Publications, Inc., New York, ninth Dover printing, (tenth GPO printing) edition, 1970.

[B+63] John Backus et al. Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 6(1):1–17, May 1963.

[BBE+10] Dario Bini, Paola Boito, Yuli Eidelman, Luca Gemignani, and Israel Gohberg. A fast implicit QR eigenvalue algorithm for companion matrices. *Linear Algebra and its Applications*, 432:2006–2032, 2010.

[Ber56] Algernon Berriman. The Babylonian quadratic equation. *The Mathematical Gazette*, 40(333):185–192, October 1956.

[BF00a] Dario Bini and Giuseppe Fiorentino. Design, analysis, and implementation of a multiprecision polynomial rootfinder. *Numerical Algorithms*, 23:127–173, 2000.

[BF00b] Dario Bini and Giuseppe Fiorentino. Numerical computation of polynomial roots using MPSolve version 2.2. Technical report, Diparti-

mento di Matematica, Università di Pisa, Via Buonarroti 2, 56127 Pisa, January 2000.

[BKS12] Ramandeep Behl, Vijay Kanwar, and Kapil Sharma. Optimal equi-scaled families of Jarratt's method. *International Journal of Computer Mathematics*, 90(2):408–422, February 2012.

[BT95] Luigi Brugnano and Donato Trigiante. Polynomial roots: the ultimate answer. *Linear Algebra and its Applications*, 225(1):207–219, August 1995.

[Car45] Girolamo Cardano. *The Rules of Algebra (Ars Magna)*. University of Pavia, Nuremberg, 1545. Translated and edited into English by T. Richard Witmer, and published by Dover (1993).

[Chu07] Changbum Chun. Some second-derivative-free variants of Chebyshev-Halley methods. *Applied Mathematics and Computation*, 191(2):410–414, August 2007.

[Col66] George Collins. PM, a system for polynomial manipulation. *Communications of the ACM*, 9(8):578–589, August 1966.

[Col71] George Collins. The SAC-1 system: An introduction and survey. In S R Petrick, editor, *SYMSAC 71: Proceedings of the second ACM Symposium on Symbolic and Algebraic Manipulation*, pages 144–152, New York, NY, USA, 1971. ACM.

[Col85] George Collins. The SAC-2 computer algebra system. In *European Conference on Computer Algebra (2)*, pages 34–35, 1985.

[Cro12] John Crowther. Anecdotes of an Atlas maintenance engineer. Website, 2012. `http://elearn.cs.man.ac.uk/anecdotes of an atlas maintenance engineer.pdf`.

[DB64] Kiril Dochev and P Byrnev. Certain modifications of Newton's method for the approximate solution of algebraic equations. *Computational Mathematics and Mathematical Physics*, 4(5):915–920, 1964.

[Dek68] Theodorus Dekker. Newton-Laguerre iteration. In *Colloque International du CNRS, Programmation en Mathématiques Numériques*, volume 165, pages 189–200, Besançon, France, 1968.

[Dun74] Donna Dunaway. Calculation of zeros of a real polynomial through factorization using Euclid's algorithm. *SIAM Journal of Numerical Analysis*, 11(6):1087–1104, December 1974.

[Dvo69] Josef Dvorčuk. Factorisation of a polynomial into quadratic factors by Newton method. *Applied Mathematics*, 14:54–80, 1969.

[Ehr67] Louis Ehrlich. A modified Newton method for polynomials. *Communications of the ACM*, 10(2):107–108, 1967.

[FB77] Brian Ford and Janet Bentley. The NAG library "machine". *ACM Signum Newsletter*, 12(4):23–24, December 1977.

[FL73] Michael Farmer and George Loizou. A note on a paper by Pólya. *BIT*, 13:8–15, 1973.

[FL75] Michael Farmer and George Loizou. A class of iteration functions for improving, simultaneously, approximations to the zeros of a polynomial. *BIT*, 15:250–258, 1975.

[FL77] Michael Farmer and George Loizou. An algorithm for the total, or partial, factorization of a polynomial. *Mathematical Proceedings of the Cambridge Philosophical Society*, 82(427):427–437, 1977.

[FL85a] Michael Farmer and George Loizou. An algorithm for the computation of zeros of a special class of entire functions. *Journal of Computational and Applied Mathematics*, 12 and 13:433–445, 1985.

[FL85b] Michael Farmer and George Loizou. Locating multiple zeros interactively. *Computers and Mathematics with Applications*, 11(6):595–603, 1985.

[FM08] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language*. O'Reilly Media, Sebastopol, CA 95472, January 2008.

[Fou10] Free Software Foundation. Gnu make manual. Website, July 2010. `http://www.gnu.org/software/make/manual`.

[Fra61] John Francis. The QR transformation, I. *The Computer Journal*, 4(3):265–271, 1961.

[Fra62] John Francis. The QR transformation, II. *The Computer Journal*, 4(4):332–345, 1962.

[GH72] Irene Gargantini and Peter Henrici. Circular arithmetic and the determination of polynomial zeros. *Numerische Mathematik*, 18:305–320, 1972.

[GJM68] G R Garside, P Jarratt, and C Mack. A new method for solving polynomial equations. *The Computer Journal*, 11(1):87–90, May 1968.

[GL86] Gene Golub and Charles Van Loan. *Matrix Computations*. North Oxford Academic, 242 Banbury Road, Oxford, OX2 7DR, England, 1986.

[GLL09] Stef Graillat, Philipe Langlois, and Nicolas Louvet. Algorithms for accurate, validated and fast polynomial evaluation. *Japan Journal of Industrial and Applied Mathematics*, 26(2-3):191–214, 2009.

[GM67] Irene Gargantini and W Münzner. An experimental program for the simultaneous determination of all zeros of a polynomial. Technical Report RZ-237, IBM Zurich Research Laboratory, 8803 Rüschlikon-ZH, Switzerland, April 1967.

[Goe94] Stefan Goedecker. Remark on algorithms to find roots of polynomials. *SIAM Journal on Scientific Computing*, 15(5):1059–1063, September 1994.

[Gra11] Törbjorn Granlund. *GNU Multiple Precision Arithmetic Library (GMP)*. Free Software Foundation, 51 Franklin Street, Boston, MA 02110-1301 USA, 5.0.2 edition, May 2011.

[Hal94] Edmond Halley. A new, exact and easy method of finding the roots of equations generally, and that without any previous reduction. *Philosophical Transactions of the Royal Society of London, Series A*, 18:136–145, 1694.

[Hen71] Peter Henrici. Circular arithmetic and the determination of polynomial zeros. In *Lecture Notes in Mathematics*, volume 228, pages 86–92. Springer-Verlag, Berlin, 1971.

[Hen74] Peter Henrici. *Applied and Computational Complex Analysis*, volume 1 of *Pure and Applied Mathematics*. Wiley-Interscience, 1974. Contents: Power Series—Integration—Conformal Mapping—Location of Zeros.

[HG69] Peter Henrici and Irene Gargantini. Uniformly convergent algorithms for the simultaneous approximation of all zeros of a polynomial. In Bruno Dejon and Peter Henrici, editors, *Proceedings of the Symposium on Constructive Aspects of the Fundamental Theorem of Algebra*, pages 77–114, London, 1969. Wiley-Interscience.

[Hou70] Alston Householder. *The Numerical Treatment of a Single Nonlinear Equation*. McGraw-Hill Inc., New York, June 1970.

[HP77] Eldon Hansen and Merrell Patrick. A family of root finding methods. *Numerische Mathematik*, 27:257–269, 1977.

[Ikh02] Monday Ikhile. On the convergence of some interval methods for simultaneous computation of polynomial zeros. *International Journal of Computer Mathematics*, 79(10):1099–1111, October 2002.

[Ili00] Anton Iliev. A generalization of Obreshkoff-Ehrlich method for multiple zeros of algebraic, trigonometric and exponential equations. *Mathematica Balkanica*, 14:17–18, 2000.

[Ish72] Misako Ishiguro. A numerical method for extraction of multiple roots of algebraic equations. *Information Processing in Japan*, 12:46–50, 1972.

[IY95] Masao Igarashi and Tjalling Ypma. Relationships between order and efficiency of a class of methods for multiple zeros of polynomials. *Journal of Computational and Applied Mathematics*, 60:101–113, 1995.

[Jar66a] Peter Jarratt. Multipoint iterative methods for solving certain equations. *The Computer Journal*, 8(4):398–400, January 1966.

[Jar66b] Peter Jarratt. Some fourth order multipoint methods for solving equations. *Mathematics of Computation*, 20(95):434–437, July 1966.

[Jar69] Peter Jarratt. Some efficient fourth-order multipoint methods for solving equations. *BIT*, 9:119–124, 1969.

[Jen69] Michael Jenkins. *Three-stage Variable-shift Iterations for the Solution of Polynomial Equations with A Posteriori Error Bounds for the Zeros*. PhD thesis, School of Computer Science, Stanford University, Stanford, California, 1969.

[JT70] Michael Jenkins and Joseph Traub. A three-stage algorithm for real polynomials using quadratic iteration. *SIAM Journal of Numerical Analysis*, 7(4):545–566, 1970.

[JT72] Michael Jenkins and Joseph Traub. Algorithm 419: Zeros of a complex polynomial. *Communications of the ACM*, 15(2):97–99, February 1972.

[JT75] Michael Jenkins and Joseph Traub. Principles for testing polynomial zerofinding programs. *ACM Transactions on Mathematical Software*, 1(1):26–34, March 1975.

[JT09] Tomas Johnson and Warwick Tucker. On a fast and accurate method to enclose all zeros of an analytic function on a triangulated domain.

Journal of Computational and Applied Mathematics, 228:418–423, 2009.

[Kah72] William Kahan. Conserving confluence curbs ill-condition. Technical Report 6, Berkley, University of California, 1972.

[Ker66] I O Kerner. Ein gesamtschrittverfahren zur berechnung der nullstellen von polynomen. *Numerische Mathematik*, 8:290–294, 1966.

[KH99] Peter Kravanja and Ann Haegemans. A modification of Newton's method for analytic mappings having multiple zeros. *Computing*, 62:129–145, 1999.

[Kim12] Young Kim. A triparametric family of three-step optimal eight-order methods for solving nonlinear equations. *International Journal of Computer Mathematics*, 89(8):1051–1059, 2012.

[Kin83] Richard King. Improving the Van de Vel root-finding method. *Computing*, 30:373–378, 1983.

[Kis54] I Kiss. Über eine verallgemeinerung des newtonschen näherungsverfahrens. *Zeitschrift für Agnewandte Mathematik und Mechanik*, 34(1):68–69, January/February 1954.

[KL07] Jisheng Kon and Yitian Li. An improvement of the Jarratt method. *Applied Mathematics and Computation*, 189(2):1816–1821, June 2007.

[KL11] Sanjay Khattri and Torgrim Log. Constructing third-order derivative-free iterative methods. *International Journal of Computer Mathematics*, 88(7):1509–1518, March 2011.

[Knu84] Donald Knuth. *The TEXBook (Computers Titles and Typesetting)*, volume A. Addison Wesley Publishing Company, Inc., 1984.

[Kuc64] Marek Kuczma. Note on Schröder's functional equation. *Journal of the Australian Mathematical Society*, 4:149–151, 1964.

[KV81] Edayatha Krishnamurthy and Hari Venkateswaran. A parallel Wilf algorithm for complex zeros of a polynomial. *BIT*, 21(4):104–111, December 1981.

[Lag66] Jean-Louis Lagouanelle. Sur une méthode de calcul de l'ordre de multiplicité des zéros d'un polynôme. *Comptes Rendus de l'Académie des Sciences, Série A*, 262(11):626–627, March 1966.

[Lam94] Leslie Lamport. *LATEX: A Document Preparation System*. Addison-Wesley Publishing Company, Inc., 1994.

[Leh61] D H Lehmer. A machine method for solving polynomial equations. *Journal of the ACM*, 8(2):151–162, April 1961.

[LNZ08] Bin Li, Jiawang Nie, and Lihong Zhi. Approximate GCDs of polynomials and sparse SOS relaxations. *Theoretical Computer Science*, 209:200–210, 2008.

[Loi83] George Loizou. Higher-order iteration functions for simultaneously approximating polynomial zeros. *International Journal of Computer Mathematics*, 14:45–58, 1983.

[Lut08] Mark Lutz. *Learning Python*. O'Reilly Media, Sebastopol, CA 95472, 4th edition, September 2008.

[Mae54] Hans J Maehly. Zur iterativen auflösung algebraischer gleichungen. *Zeitschrift für Angewandte Mathematik und Physik*, 5(3):260–263, May 1954.

[Mar66] Morris Marden. *Geometry of Polynomials (Mathematical Surveys Number 3)*. American Mathematical Society, Providence, Rhode Island, 2nd edition, 1966.

[Mat12] Matlab. *Version 8.0.0.783 (R2012b)*. The MathWorks Inc., Natick, Massachusetts, USA, August 2012.

[Mat13] Matlab. Matlab. the language of technical computing. Webpage, 2013. `http://www.mathworks.co.uk/products/matlab`.

[McC61] Daniel McCracken. *A Guide to Fortran Programming*. John Wiley & Sons Inc., New York, USA, December 1961.

[Mit83] Taketomo Mitsui. A graphical technique for nonlinear algebraic equations. *International Journal of Computer Mathematics*, 13:245–261, 1983.

[Miy93] Tsuyako Miyakoda. Multiplicity estimating algorithm for zeros of a complex polynomial and its applications. *Journal of Computational and Applied Mathematics*, 46(3):357–368, 1993.

[Mon12] Mansor Monsi. The point symmetric single-step procedure for the simultaneous approximation of polynomial zeros. *Malysian Journal of Mathematical Science*, 6(1):29–46, 2012.

[MP83] Gradimir Milovanović and Miodrag Petković. On the convergence order of a modified method for simultaneous finding polynomial zeros. *Computing*, 30:171–178, 1983.

[MV95] Fadi Malek and Rémi Vaillancourt. Polynomial zerofinding iterative matrix algorithms. *Computers and Mathematics with Applications*, 29(1):1–13, 1995.

[MW08] Shi-Mei Ma and Yi Wang. q-Eulerian polynomials and polynomials with only real zeros. *The Electronic Journal of Combinatorics*, 15(R17):1–9, 2008.

[Neu88] Arnold Neumaier. An existence test for root clusters and multiple roots. *Zeitschrift für Angewandte Mathematik und Mechanik*, 68(6):256–257, 1988.

[New36] Isaac Newton. *Method of Fluxions*. Henry Woodfall, London, 1736.

[Olv52] Frank Olver. Evaluation of zeros of high degree polynomials. *Philosophical Transactions of the Royal Society of London, Series A*, 244(885):385–415, 1952.

[Ost66] Alexander Ostrowski. *Solution of Equations and Systems of Equations*. Academic Press Inc., New York, 2nd edition, 1966.

[Pan97] Victor Pan. Solving a polynomial equation: Some history and recent progress. *SIAM Review*, 39(2):187–220, June 1997.

[Pet80] Miodrag Petković. *Some Iterative Interval Methods for Solving Equations*. PhD thesis, University of Niš, 1980.

[Pet81] Miodrag Petković. On a generalisation of the root iterations for polynomial complex zeros in circular interval arithmetic. *Computing*, 27:37–55, 1981.

[Pet82] Miodrag Petković. Generalised root iterations for the simultaneous determination of multiple complex zeros. *Zeitschrift für Angewandte Mathematik und Mechanik*, 62:627–630, 1982.

[Pet86] Ljiljana Petković. A note on the evaluation in circular arithmetic. *Zeitschrift für Angewandte Mathematik und Mechanik*, 66(8):371–373, 1986.

[Pet89] Miodrag Petković. Iterative methods for simultaneous inclusion of polynomial zeros. In A Dold and B Eckmann, editors, *Lecture Notes in Mathematics*, volume 1387. Springer-Verlag, Berlin, 1989.

[Pet90] Miodrag Petković. Schröder-like algorithms for multiple complex zeros of a polynomial. *Computing*, 45:39–50, 1990.

[Pin76] James Pinkert. An exact method for finding the roots of a complex polynomial. *ACM Transactions on Mathematical Software*, 2:351–363, December 1976.

[PM06] Miodrag Petković and D M Milosevic. On a new family of simultaneous methods with corrections for the inclusion of polynomial zeros. *International Journal of Computer Mathematics*, 83(3):299–317, 2006.

[PM12] Miodrag Petković and Mimica Milošević. Efficient Halley-like methods for the inclusion of multiple zeros of polynomials. *Computational Methods in Applied Mathematics*, 12(3):69–81, 2012.

[PMP10] Miodrag Petković, Dušan Milošević, and Ivan Petković. On the improved Newton-like methods for the inclusion of polynomial zeros. *International Journal of Computer Mathematics*, 87(8):1726–1735, 2010.

[Pom71] Tomaso Pomentale. A class of iterative methods for holomorphic functions. *Numerische Mathematik*, 18:193–203, 1971.

[PPS89] Miodrag Petković, Ljiljana Petković, and Lidija Stefanović. On the R-order of a class of simultaneous iterative processes. *Zeitschrift für Angewandte Mathematik und Mechanik*, 69:199–201, 1989.

[PPŽ03] Ljiljana Petković, Miodrag Petković, and Dragan Živković. Hansen-Patrick's family is of Laguerre type 1. *Novi Sad Journal of Mathematics*, 33(1):109–115, 2003.

[Pre71] Marica Prešić. Un procédé itératif pour déterminer k zéros d'un polynôme. *Comptes Rendus de l'Académie des Sciences, Série A*, 273:446–449, Septembre 1971.

[Pre73] Marica Prešić. A certain iterative method for the simultaneous determination of the real solution of an equation. *Matematički Vesnik*, 10(25):299–308, 1973.

[PRM12] Miodrag Petković, Lidija Rančić, and Mimica Milošević. On the improved Farmer-Loizou method for finding polynomial zeros. *International Journal of Computer Mathematics*, 89(4):499–509, 2012.

[Ral66] Louis Rall. Convergence of the Newton process to multiple solutions. *Numerische Mathematik*, 9:23–37, 1966.

[Sch70] Ernst Schröder. Über unendlich viele algorithmen zur auflösung der gleichungen. *Mathematische Annalen*, 2:317–365, 1870.

[Sha07] J R Sharma. A family of improved Jarratt multipoint methods. *International Journal of Computer Mathematics*, 84(7):1027–1034, 2007.

[SP82] Lidija Stefanović and Miodrag Petković. On the simultaneous improving K inclusive disks for polynomial complex zeros. *Freiburger Intervall-Berichte*, 7:1–13, 1982.

[Ste69] Gilbert Stewart. On Lehmer's method for finding the zeros of a polynomial. *Mathematics of Computation*, 23:829–835, 1969.

[Sti72] Iain Stinton. *The Story of ATLAS, a Computer*. Richard Williams and Partners, PO Box 8, Llandudno, Wales, Great Britain, 1972.

[SXL09] Li Shengguo, Liao Xiangke, and Cheng Lizhi. A new fourth-order iterative method for finding multiple roots of nonlinear equations. *Applied Mathematics and Computation*, 215:1288–1292, 2009.

[Tra64] Joseph Traub. *Iterative Methods for the Solution of Equations*. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1964.

[Uhl99] Frank Uhlig. General polynomial roots and their multiplicities in $O(n)$ memory and $O(n^2)$ time. *Linear and Multilinear Algebra*, 46:327–359, 1999.

[Ver11] Jan Verschelde. Multiple roots and approximate GCDs. Technical report, Department of Mathematics, Statistics and Computer Science, University of Illinois at Chicago, 2011. MCS 563, Lecture 20.

[vzGG99] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, first edition, 1999.

[Wei03] Karl Weierstrass. Neuer beweis des satzes, dass jede ganze rationale funktion einer veränderlichen dargestellt werden kann als ein produkt aus linearen funktionen derselben veränderlichen. *Gesammelte Werke*, 3:251–269, 1903.

[Wel13] Paul Wellin. *Programming with Mathematica, An Introduction*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, first edition, 2013.

[WH10] Joab Winkler and Madina Hasan. A non-linear structure preserving matrix method for the low rank approximation of the Sylvester resultant matrix. *Journal of Computational and Applied Mathematics*, 234:3226–3242, 2010.

[WH13] Joab Winkler and Madina Hasan. An improved non-linear method for the computation of a structured low rank approximation of the Sylvester resultant matrix. *Journal of Computational and Applied Mathematics*, 237:253–268, 2013.

[WHL12] Joab Winkler, Madina Hasan, and Xin Lao. Two methods for the calculation of the degree of an approximate greatest common divisor of two inexact polynomials. *Calcolo*, 49:241–267, 2012.

[Wil59a] James Wilkinson. The evaluation of the zeros of ill-conditioned polynomials. Part I. *Numerische Mathematik*, 1:150–166, 1959.

[Wil59b] James Wilkinson. The evaluation of the zeros of ill-conditioned polynomials. Part II. *Numerische Mathematik*, 1:167–180, 1959.

[Wil78] Herbert Wilf. A global bisection algorithm for computing the zeros of polynomials in the complex plane. *Journal of the ACM*, 25(3):415–420, 1978.

[Win07] Joab Winkler. Polynomial roots and approximate greatest common divisors. In Joab Winkler, editor, *Solving Polynomial Equations: A New Approach*. University of Oxford, September 2007.

[Win11] Joab Winkler. The computation of multiple roots of polynomials whose coefficients are inexact. In *Geometry Seminar*. New York University, November 2011.

[WL11] Joab Winkler and Xin Lao. The calculation of the degree of an approximate greatest common divisor of two polynomials. *Journal of Computational and Applied Mathematics*, 235:1587–1603, 2011.

[WLH12] Joab Winkler, Xin Lao, and Madina Hasan. The computation of multiple roots of a polynomial. *Journal of Computational and Applied Mathematics*, 236:3478–3497, 2012.

[Zen04] Zhonggang Zeng. Algorithm 835: MULTROOT — A Matlab package for computing polynomial roots and multiplicities. *ACM Transactions on Mathematical Software*, 30(2):218–236, June 2004.

[Zen05] Zhonggang Zeng. Computing multiple zeros of inexact polynomials. *Mathematics of Computation*, 74(250):869–903, July 2005.

[Zen09] Zhonggang Zeng. The approximate irreducible factorization of a univariate polynomial revisited. In Jeremy Johnson, Hyungju Park, and Erich Kaltofen, editors, *Proceedings of the 2009 International Sympo-*

sium on Symbolic and Algebraic Computation, (ISSAC 2009), pages 367–374, New York, NY, USA, July 2009. ACM.

[Zen12] Zhonggang Zeng. NAClab 2.0 – Numerical Algebraic Computing toolbox for Matlab. Website, September 2012. `http://www.neiu.edu/~zzeng/NAClab.html`.