Complex Root Tracking In Dispersion Relations Arising From Non Self-Adjoint Boundary Value Problems

Author

Louis Runcieman

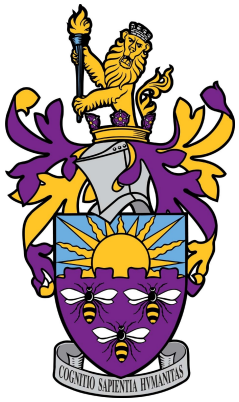
ID: 8442265

Supervisor

Matthias Heil

Year of submission

2016



A dissertation submitted to the University of Manchester
for the degree of
Master of Science in Applied Mathematics with Industrial Modelling
in the faculty of
Engineering and Physical Sciences, School of Mathematics

Contents

List of Figures

List of Tables

Preface

Abstract

Dispersion relations arising from non self-adjoint boundary value problems have complex roots, or eigenvalues. When constructing accurate truncated modal solutions it is essential all significant eigenvalues and their corresponding modes have been determined. After a truncated modal solution has converged validity can be confirmed by comparison with boundary data.

An example problem, discussed in this dissertation, is determining the time harmonic pressure in a two dimensional semi-infinite duct lined with a sound absorbent material. The dispersion relation depends on the complex-valued wall admittance of the duct's lining. For vanishing wall admittance (hard-walled) the problem is self-adjoint and has real eigenvalues. For finite-valued wall admittance the problem is non self-adjoint and has complex eigenvalues.

To ensure all significant eigenvalues have been determined in the non self-adjoint problem we extend an existing complex root finder to allow root tracking in response to changes in a control parameter. The newly developed methodology is used to track the eigenvalues of the hard-walled self-adjoint problem whilst wall admittance is incremented to form the non self-adjoint problem.

Declaration

No portion of the work referred to in the dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Intellectual property statement

i. The author of this dissertation (including any appendices and/or schedules to this dissertation) owns certain copyright or related rights in it (the Copyright) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this dissertation, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has entered into. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the Intellectual Property) and any reproductions of copyright works in the dissertation, for example graphs and tables (Reproductions), which may be described in this dissertation, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this dissertation, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see `http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=487`), in any relevant Dissertation restriction declarations deposited in the University Library, The University Librarys regulations (see `http://www.manchester.ac.uk/library/ab-outus/regulations`) and in The Universitys Guidance on Presentation of Dissertations.

Acknowledgements

Many thanks to my supervisor Matthias Heil who always had his door open with great patience towards my data handling. Further thanks to Natasha Willoughby and Phil Cotterill from Thales UK for conceiving, giving application to and funding this dissertation project.

1 Introduction

1.1 Motivation and goals

The propagation of waves through ducts or pipes, known as waveguides, has long been of physical and mathematical interest. The study of such a problem predates Lord Rayleigh, who wrote one of the first complete works on the mathematical treatment of acoustics, *Theory of Sound* (1877) [10].

Amongst many other significant findings, Rayleigh determined modal solutions for infinitely long rectangular and circular waveguides, which provide ideal channels for noise transmission. Such problems are still of great physical interest today and have applications primarily in the reduction of noise in aircraft engines and industrial heating, ventilation and air conditioning (HVAC) systems [8].

Waveguides can be lined with sound absorbent materials to suppress noise, providing acoustic impedance boundary conditions. Acoustic impedance of a material is measured experimentally using methods such as the two microphone method and describes the opposition a material has to an acoustic flow [5]. It is unrealistic to consequently determine the effect different acoustic impedances have on noise reduction for complex and likely expensive systems; we may, however, proceed by the concise language of mathematics. Thales, the company proposing this dissertation, are interested in such problems.

A representative problem supplied by Thales (detailed in Section 2) is a two dimensional semi-infinite duct lined with a sound absorbent material and forced by a time harmonic oscillator at the base of the duct [1]. This problem is governed by the Helmholtz equation, for which separable solutions exist. Accordingly, the problem reduces to determining roots of a governing dispersion relation which must be satisfied to ensure non-trivial solutions. The sound absorbent lining removes conservation from the system, mathematically meaning the problem is non self-adjoint. Consequently the roots, or eigenvalues, of the governing dispersion relation may take

complex values which can be difficult to analytically determine. In 2014 Franklin successfully completed a master's dissertation proposed by Thales which addressed this complication [4]. Franklin developed MATLAB code which determines all roots of a complex function $f(z)$ in a specified complex region.

In practice Thales found that dispersion relations of interest produce unusual root behaviour for changing values in wall admittance (which is inversely proportional to wall impedance) [13]. Consequently some eigenvalues were being missed; the corresponding modes were not present in the truncated modal solution producing inaccuracy. This was confirmed by Thales by noting that as an eigenvalue leaves a fixed complex search region for a change in value to the wall admittance the truncated modal solution no longer accurately satisfies the system's boundary conditions. Furthermore, Thales encountered a problem in constructing an order to eigenvalues (due to the lack of natural ordering in complex values) and consequently determining which eigenvalues correspond to modes of greatest significance to the modal solution.

In the hard-walled case (vanishing wall admittance) the problem is self-adjoint and has non-negative real eigenvalues, which can be ordered by increasing size. Smaller eigenvalues correspond to modes of greater significance to the modal solution. Thales propose that this order can be exploited in the non self-adjoint case by tracking eigenvalues of the self-adjoint problem as the wall admittance is incremented to reform the non self-adjoint problem. Thus concluding the motivation for the dissertation's goals set by Thales:

- improve an existing complex root finder (secondary goal);
- extend the root finder to allow root tracking in response to changes in an implicit parameter (primary goal);
- check for a specific test case[1] that root tracking suffices to obtain accuracy in the truncated modal solution (tertiary goal).

[1]Noting that limited mathematical theory surrounds non self-adjoint problems and the results may not be generalisable.

1.2 Outline

Section 2 details the aforementioned two dimensional semi infinite lined duct problem presented by Thales including the derivation of the modal solution and governing dispersion relation whose roots will need to be tracked for changing values of wall admittance.

Section provides the theory Franklin [4] implements in MATLAB to find roots of complex functions in a given complex region. Limitations of Franklin's code are given in Section 3.2.1. The improved implementation is detailed by a list of updates and description of the improved file structure in Sections 3.3.1 and 3.3.2 respectively. A user guide is given for the updated root finder in Section 3.3.3.

Section 4 describes the application of the complex root finder to allow root tracking as a parameter is incremented. Section 4.1 describes the developmental process and theory required to achieve this goal and Section 4.2 describes the theory's implementation in MATLAB including the procedures set in place to ensure roots are effectively tracked. A user manual detailing all functionality and features is given with examples in Section 4.4.

Finally Section 5 provides two test cases for the representative problem described in Section 2. In both cases, a truncated modal solution is first determined by searching for roots of the governing dispersion relation in a fixed complex region; convergence is tested for different numbers of modes and once sufficiently converged, the solution is shown to contradict the system's boundary condition at the base of the duct. Following this, roots of the governing dispersion relation are determined by a root tracking method. The truncated modal solution formed from this method is correctly verified against the boundary condition at the base of the duct before comparing it to the inaccurate modal solution obtained without root tracking.

2 Representative Non Self-adjoint Problem

We now consider the problem presented by Thales [1]. The governing equations will be derived before obtaining a modal solution and the corresponding dispersion relation.

2.1 Derivation of governing equations

Consider a two dimensional semi-infinite fluid filled duct with geometry $x \in [-1, 1]$ and $z > 0$. Suppose walls of the duct are lined with a sound absorbent material of acoustic impedance $Z_{\pm}$ at $x = \pm 1$ and at they base of the duct there is a time harmonic oscillating plate. The following figure details the system.

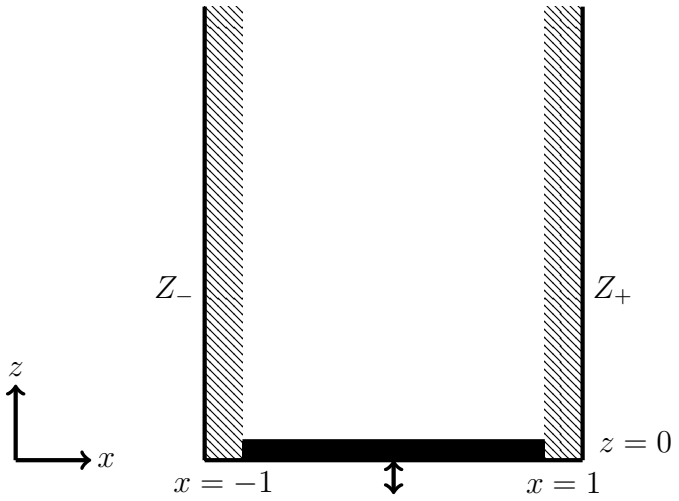


Figure 2.1: Semi-infinite rectangular duct with absorbent lining and oscillating base

For spatial coordinate $\mathbf{x} = (x, z)$ and time t, a linear fluid pressure perturbation $p(\mathbf{x}, t)$ in an acoustic waveguide satisfies the linear acoustic pressure wave equation and the linearised Euler's equation [2]. That is,

$$\nabla^2 p - \frac{1}{c^2} \frac{\mathrm{d}^2 p}{\mathrm{d}t^2} = 0, \tag{2.1}$$

for acoustic speed of sound c (in medium) and

$$\frac{\partial \mathbf{u}}{\partial t} = -\frac{\nabla p}{\rho}, \tag{2.2}$$

for ambient density ρ and linear velocity perturbation $\mathbf{u}(\mathbf{x}, t)$.

Restrict attention to time harmonic solutions of angular frequency ω by applying separation of variables to the time dependency:

$$p(\mathbf{x}, t) = P(\mathbf{x})T(t).$$

Substituting this into (2.1) gives

$$\frac{\nabla^2 P}{P} - \frac{1}{c^2}\frac{T''}{T} = 0,$$

after dividing by p (assuming non-triviality). Therefore

$$\frac{\nabla^2 P}{P} = \frac{1}{c^2}\frac{T''}{T} = -\kappa^2, \tag{2.3}$$

for separation constant $\kappa > 0$ known as the wavenumber. This arises since the left hand side is purely spatial and the right hand side purely time dependent. Hence, for angular frequency ω satisfying $\omega^2 = c^2\kappa^2$,

$$T'' = -\omega^2 T,$$

with solution $T(t) = e^{-i\omega t}$, chosen with negative sign only by convention[2]. Therefore

$$p(\mathbf{x}, t) = P(\mathbf{x})e^{-i\omega t}.$$

Substituting this into (2.1) gives

$$\left(\nabla^2 P + \frac{\omega^2}{c^2}P\right) e^{-i\omega t} = 0,$$

[2]One may choose (2.3) to have a positive right hand side; this would yield the same physical solution as the sign change is equivalent to mapping time $t \to -t$.

thus deriving the Helmholtz equation[3]

$$\nabla^2 P + \kappa^2 P = 0, \tag{2.4}$$

for wavenumber κ defined by $\kappa^2 = \omega^2/c^2$.

Separable solutions can also exist for the Helmholtz equation, reducing (2.4) into a system of ordinary differential equations. Combining the Helmholtz equation with boundary conditions to form a boundary value problem (BVP) can allow the pressure to be expressed as an expansion of eigenfunctions, known as a modal solution. Modal solutions exist for systems of harmonic motion. They are constructed by the typically infinite summation of eigenfunctions, which are known as (normal) modes of the system. Modes are described as normal since they evolve independently, meaning the excitation of a single mode will affect only that mode.

We now define our boundary conditions. Acoustic impedance, Z, is an intrinsic medium property defined as the ratio of acoustic pressure to acoustic normal velocity [6]. Let $u_n = \mathbf{u} \cdot \widehat{\mathbf{n}}$ for unit norm $\widehat{\mathbf{n}}$. Then the acoustic impedance is measured at the insulated walls, $x = \pm 1$, giving

$$Z_\pm = \frac{p(x = \pm 1)}{u_n(x = \pm 1)}, \tag{2.5}$$

where Z_- and Z_+ is the acoustic impedance at the left and right wall respectively. Clearly at $x = \pm 1$ we have $\widehat{\mathbf{n}} = (\pm 1, 0)$, thus

$$u_n(x = \pm 1) = \pm u_x, \tag{2.6}$$

where u_x is the x-component of the acoustic velocity.

Taking the x-component of the linearised Euler's equation (2.2) and rearranging gives

$$\frac{\partial u_x}{\partial t} = -\frac{1}{\rho}\frac{\partial p}{\partial x}\,.$$

[3]One may also derive the Helmholtz equation by applying a Fourier transformation in the time variable t to the wave equation.

Therefore, assuming time harmonic solutions,

$$-i\omega U_x e^{-i\omega t} = -\frac{1}{\rho}\frac{\partial P}{\partial x}\, e^{-i\omega t},$$

where U_x is the x-component of the time harmonic velocity vector. This rearranges to

$$U_x = \frac{1}{i\rho\omega}\frac{\partial P}{\partial x}\,, \tag{2.7}$$

after cancelling the non-zero exponential factor.

Substitution of (2.7) into (2.5) (in time harmonic form) gives

$$\frac{\partial P}{\partial x} = \pm\frac{i\rho\omega}{Z_\pm}P, \quad \text{at} \quad x = \pm 1, \tag{2.8}$$

after rearranging and recalling (2.6). Now we may relate the wall impedance to the inversely proportional wall admittance [1], S, defined as

$$Z = \frac{i\rho\omega}{S}.$$

Substituting this into (2.8) allows the formulation of the following Robin boundary condition

$$\frac{\partial P}{\partial x} = \pm S_\pm P, \quad \text{at} \quad x = \pm 1, \tag{2.9}$$

continuing the $\pm$ notation such that S_- and S_+ is the wall admittance at the left and right wall respectively. In mind of simplicity we will consider a symmetric duct, meaning $S_+ = S_- = S$. Further to this we will assume S is independent of z and the wall lining material is such that no sound propagation exists in the material parallel to the wall. Such a material is known as *locally reacting*, meaning the impedance is independent of the angle of incidence - allowing us to assume an approximately zero transmission angle [11].

By taking the z component of the linearised Euler's equation and assuming the oscillating plate has velocity profile $f(x)$ we have, by a similar derivation for (2.9), the following Neumann condition

$$\frac{\partial P}{\partial z} = i\kappa f(x), \quad \text{at} \quad z = 0.$$

Finally, by physical reasoning, pressure is atmospheric at $z \to \infty$, which we will assume to be vanishing.

In summary our time harmonic pressure P satisfies the following non self-adjoint BVP:

$$\begin{cases} \nabla^2 P + \kappa^2 P = 0, & \text{for } -1 < x < 1 \text{ and } z > 0, \quad (2.10a) \\ \dfrac{\partial P}{\partial x} = \pm SP, & \text{at } x = \pm 1, \quad (2.10b) \\ \dfrac{\partial P}{\partial z} = i\kappa f(x), & \text{at } z = 0. \quad (2.10c) \\ P \to 0, & \text{for } z \to \infty. \quad (2.10d) \end{cases}$$

2.2 Derivation of modal solution and dispersion relation

Consider modal solutions of form $P(x,z) = X(x)Z(z)$ to (2.10a), giving

$$X''Z + XZ'' + \kappa^2 XZ = 0,$$

which implies

$$\frac{X''}{X} = -\frac{Z''}{Z} - \kappa^2 = \widehat{\lambda}, \quad (2.11)$$

for separation constant $\widehat{\lambda}$. Note that since BVP (2.10) is non self-adjoint we expect $\widehat{\lambda}$ to take complex values. We may therefore let, for convenience, $\widehat{\lambda} = -\lambda^2$ for some $\lambda \in \mathbb{C}$ without loss of generality.

Solving (2.11) gives

$$X(x) = A\cos\lambda x + B\sin\lambda x,$$
$$Z(z) = Ce^{i\xi z},$$

for $\xi = \sqrt{\kappa^2 - \lambda^2}$. By boundary condition (2.10d) we must not have exponentially growing terms as $z \to \infty$. Hence, if ξ is complex-valued (non-zero imaginary part) then the sign of the square root must be taken to ensure the imaginary part is positive.

By an ansatz and for ease of calculation let us decompose the modal solution into the sum of symmetric and antisymmetric modes for which we anticipate a symmetric and antisymmetric dispersion relation. For clarity we will label the eigenvalues of symmetric and antisymmetric modes by μ and ν respectively. That is

$$P(x,z) = \sum_{n=1}^{\infty} D_n \cos(\mu_n x) e^{i\widehat{\xi}_n z} + \sum_{m=1}^{\infty} E_n \sin(\nu_m x) e^{i\check{\xi}_m z}, \tag{2.12}$$

such that

$$\widehat{\xi} = \sqrt{\kappa^2 - \mu^2},$$
$$\check{\xi} = \sqrt{\kappa^2 - \nu^2},$$

where signs are taken to ensure positive imaginary parts in both cases.

Let us only consider symmetric $f(x)$ in the range $x \in [-1, 1]$. Consequently, the duct is transversely symmetric. This further simplifies our problem by allowing us to deduce that $E_m = 0$ for all m.

Boundary condition (2.10c) can be applied to (2.12) to determine the coefficients D_n. We have

$$\kappa f(x) = \sum_{n=1}^{\infty} D_n \xi_n \cos(\mu_n x). \tag{2.13}$$

where the hat on $\widehat{\xi}_n$ has been dropped for ease.

Orthogonality of complex eigenfunctions arising from non self-adjoint BVPs is not a trivial property and cannot be assumed. We instead proceed by truncating (2.12) and by extension (2.13) to N terms; we can assume equality for N sufficiently large. From this we can determine a linear system of N equations by multiplying both sides by $\cos(\mu_p x)$ for $1 \leqslant p \leqslant N$ and integrating over the transverse range, $x \in [-1, 1]$. That is,

$$\int_{-1}^{1} \kappa f(x) \cos(\mu_p x)\, \mathrm{d}x = \sum_{n=1}^{N} \left[D_n \xi_n \int_{-1}^{1} \cos(\mu_n x) \cos(\mu_p x)\, \mathrm{d}x \right], \tag{2.14}$$

after swapping the order of summation and integration. For each value of p we have a

unique linear equation. For ease, let us write (2.14) in matrix form by denoting

$$f_p = \kappa \int_{-1}^{1} f(x) \cos\left(\mu_p x\right) \mathrm{d}x,$$
$$\mathrm{M}_{np} = \xi_n \int_{-1}^{1} \cos\left(\mu_n x\right) \cos\left(\mu_p x\right) \mathrm{d}x,$$

reducing (2.14) to

$$\mathbf{f} = \mathrm{M}\mathbf{d},$$

where $\mathbf{f}$ is a length N vector with ith entry f_i, M is a $N \times N$ matrix with (i, j)th entry M_{ij} and $\mathbf{d}$ is a length N vector with ith entry D_i. We can determine $\mathbf{d}$ by the method of least squares [9].

Finally we must determine the governing dispersion relation; we apply boundary conditions (2.10b) to a general symmetric eigenfunction $\phi_n(x, z)$. That is,

$$\frac{\partial \phi_n}{\partial x} = \pm S \phi_n \quad \text{at} \quad x = \pm 1,$$

where

$$\phi_n(x, z) = D_n \cos\left(\mu_n x\right) e^{i\xi_n z}.$$

Specifically,

$$-D\mu_n \sin\left(\mu_n x\right) e^{i\xi_n z} = \pm SD \cos\left(\mu_n x\right) e^{i\xi_n z} \quad \text{at} \quad x = \pm 1,$$

which, after assuming non-triviality ($D \neq 0$), reduces to the dispersion relation

$$\mu_n \sin \mu_n + S \cos \mu_n = 0, \tag{2.15}$$

with roots μ_n.

The roots of the dispersion relation (2.15) take complex values. We consider now Wilf's quadsection algorithm [12], a method which Franklin implemented into MATLAB to determine the roots of complex functions in a complex region [4].

Following this, we will consider a method which utilises the code written by Franklin to ensure all eigenvalues of significance are determined. We extend the root

finder to allow root tracking in response to changes in a control parameter. The newly developed methodology is used to track the eigenvalues of the hard-walled self-adjoint problem ($S = 0$) as the wall admittance is incremented to form the non self-adjoint problem.

3 Finding Complex Roots

In this section we will describe the algorithm Franklin implemented in MATLAB to determine all roots of complex functions $f(z)$ in a specified complex region before describing drawbacks of the implementation. Following this, we discuss improvements made and give a user manual to the updated code.

3.1 Wilf's quadsection algorithm

Wilf [12] suggested a simple algorithm for determining the roots of a complex function $f = f(z)$ in a region R of the complex plane. It relies on being able to calculate the number of roots within the region. Once this is known, the region is quadsectioned into four subregions, the number of roots is recalculated and then each subregion is treated recursively as region R. This continues until each subregion is sufficiently small so that the centre of a subregion can be assumed to be a root value and the number of roots in this subregion simply the multiplicity of a single root.

To proceed we must calculate the number of roots in a given region. Franklin [4] assumes f to have no poles. As a result, Cauchy's argument principle [7] can be applied.

Theorem 3.1 (Cauchy's Argument Principle)**.** *Suppose $f = f(z)$ is analytic inside a simple closed contour R except for n_p poles (strictly inside R) and suppose $f \neq 0$ on R. In addition, let n_r be the number of roots of f in R. Then*

$$N = \frac{1}{2\pi i} \oint_R \frac{f'(z)}{f(z)} \, \mathrm{d}z = n_r - n_p, \tag{3.1}$$

where N is known as the winding number (of f around R).

Since f has no poles, $n_p = 0$. Direct calculation of (3.1) yields the number of roots in R and is later referred to as the *principle argument method*.

Franklin also implemented a method of determining the winding number without requiring the calculation of f' by noting

$$\frac{f'(z)}{f(z)} = \frac{\mathrm{d}}{\mathrm{d}z}\operatorname{Log}\big(f(z)\big),$$

where Log is the principle value logarithm defined by

$$\operatorname{Log}\big(f(z)\big) = \log\big|f(z)\big| + \arg\big(f(z)\big)\, i.$$

Substituting this into (3.1) yields

$$\begin{aligned} N &= \frac{1}{2\pi i}\oint_R \frac{\mathrm{d}}{\mathrm{d}z}\operatorname{Log}\big(f(z)\big)\,\mathrm{d}z, \\ &= \frac{1}{2\pi i}\left(\oint_R \log\big|f(z)\big|\,\mathrm{d}z + i\oint_R \arg\big(f(z)\big)\,\mathrm{d}z\right). \end{aligned}$$

By definition $\log\big|f(z)\big|$ is single valued and therefore its integral over R will be zero. Hence

$$\begin{aligned} N &= \frac{1}{2\pi}\oint_R \arg\big(f(z)\big)\,\mathrm{d}z, \\ &= \frac{1}{2\pi}\Delta_R \arg\big(f(z)\big), \end{aligned} \tag{3.2}$$

where Δ_R represents the total change of its argument over the contour R. This is determined by dividing each edge of R into chords and tracking the change in value of f over the end points of each chord. A change by $\pm\pi$ increases the winding number (starting at zero) by ± 1. This method is later referred to as the *chord approach*.

Upon determining the winding number, (3.1) may be weighted with z^n for some $n \in \mathbb{N}$. We find a winding integral I_n of order n which can be written as a polynomial [3] in n_r and n_p:

$$I_n = \frac{1}{2\pi i}\oint_R z^n \frac{f'(z)}{f(z)}\,\mathrm{d}z = \sum_i (z_{r(i)})^n - \sum_j (z_{p(j)})^n, \tag{3.3}$$

where $z_{r(i)}$ and $z_{r(j)}$ are the ith roots and jth poles respectively.

Under the assumption f has no poles (3.3) reduces to a polynomial whose roots are also the roots of f. As there are only explicit solutions to polynomials of degree 4

or less, quadsectioning must be applied until each subregion has $n_r \leqslant 4$, illustrated in the following figure.

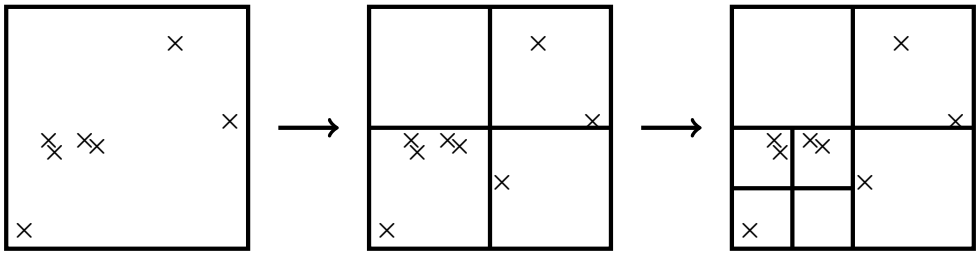


Figure 3.1: Quadsectioning region so that 4 or fewer roots $\times$ are in each subregion

For a single root ($n_r = 1$) we have

$$I_1 = z_{r(1)}.$$

For $n_r = 2$ we have

$$I_1 = z_{r(1)} + z_{r(2)},$$
$$I_2 = (z_{r(1)})^2 + (z_{r(2)})^2,$$

which can be rearranged to a single second order polynomial:

$$z^2 - I_1 z + \frac{1}{2}(I_1^2 - I_2) = 0,$$

whose roots are $z_{r(1)}$ and $z_{r(2)}$. Similarly for $n_r = 3$ and $n_r = 4$ we have

$$z^3 - I_1 z^2 + \frac{1}{2}(I_1^2 - I_2)z + \frac{I_1 I_2}{6} - \frac{I_1^3}{6} - \frac{I_3}{3} = 0,$$

and

$$z^4 - I_1 z^3 + \frac{1}{2}(I_1^2 - I_2)z^2 + \frac{1}{2}\left(I_1 I_2 - \frac{1}{3}I_1^3 - \frac{2}{3}I_3\right)z + \frac{1}{24}(I_1^4 - 6I_1^2 I_2 + 3I_2^2 + 8I_1 I_3 - 6I_4) = 0,$$

respectively. Thus, under the assumption f has no poles in R and is analytic we can calculate the roots (and their multiplicities) of f.

3.2 Existing root finder

Franklin [4] implemented Wilf's Quadsection method in MATLAB, creating the function `dispersionroot`. This function determines all roots of a complex-valued function $f = f(z)$ in a specified complex region R. Two methods are available for determining the winding number of a function f around R. Method 0, the argument principle method, calculates the winding number by computing (3.1) and method 1, the chord approach, calculates the winding number by computing (3.2).

To run this program MATLAB's Symbolic Math Toolbox is required. Top level control is entirely through `dispersionroot`. The additional scripts required to run this program (located in the same directory as `dispersionroot`) are given in the following table, provided for later reference.

`WindNum`
`ArgWN`
`ArgWNSide`
`Quadsec`
`WNContourPA`
`WNChordContour`
`WindIntPoly`

Table 1: Additional scripts required to run `dispersionroot`

Franklin implemented two checks to ensure the winding number is correctly calculated. If the winding number is non-integer Wilf suggests a root is on or near the boundary of the search region [12]; in such a case, `dispersionroot` increases the region by 1% in every direction. This is done recursively until the winding number is integer and is depicted in the following figure.

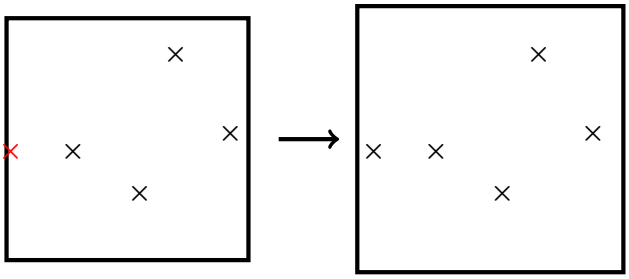


Figure 3.2: Resizing region due to root × located too close to boundary

If the winding number for a subregion is non-integer after quadsectioning then it is assumed a root is on or near an inner boundary. In such a case the centre of the quadsection is moved. In particular, the centre is moved by 1% of the length of the real limits in the increasing real direction and 1% of the length of the imaginary limits in the increasing imaginary direction. Each time the winding number is recalculated and this is applied recursively until the winding number is integer. If the centre point has has been moved more than 10 times the function outputs an error and breaks. This check also applies to subregions which undergo quadsectioning and is depicted in the following figure.

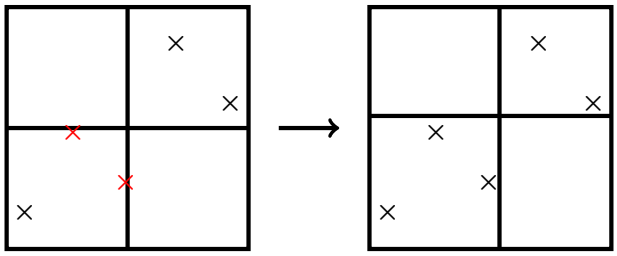


Figure 3.3: Relocating quadsection centre due to roots × too close to boundaries

Furthermore, if the region has been quadsectioned more than 20 times the user has a choice to either continue quadsectioning, apply the Newton-Raphson method to the centre of the subregion or assume the centre point of the subregion is a single root with multiplicity equal to the subregion's winding number.

3.2.1 Limitations and drawbacks

- If method 1 has been chosen and we find a root on the region boundary then the region is increased by 1% in every direction. The winding number is then incorrectly recalculated by `WindNum`, the method 0 function. This is fixed in later versions.

- Roots of functions of the form $f(z) = az^b$ for any $a \in \mathbb{C}$ and non-negative integer b cannot be computed by `dispersionroot`. The error origins from the MATLAB function `quadgk`. Some testing implies this error occurs when determining the winding integral, which has integrand $z^n f'/f$. As values of n are cycled the integrand will become a non-symbolic scalar which `quadgk` cannot integrate over. This problem is MATLAB internal and MathWorks have been notified.

- This program makes no use of MATLAB's variable input and output parameters. The winding number is output and saved to memory, which realistically is rarely useful. If method 0 is chosen, `numchords` need not be specified. In later versions method 0 is chosen as default and method 1 is chosen by specifying `'chords'` as an optional input. This is since method 0 has no difficulty determining the winding number in the vast majority of cases and method 1 is much more computationally expensive than method 0.

- The program has no error checking for inputs. When inputs are incorrectly given (for example choosing an empty search region `xsides = [2, 2]`) MATLAB outputs an internal error

```
Output argument "r" (and maybe others) not assigned during
    call to "dispersionroot".
```

  meaning the user is not given a comprehensible reason why the program fails to find roots. This error is also given when a winding number `WN` is in the range $[0, 1)$.

- Throughout the program variable and function names are ambiguous. For example, the main function `dispersionroot` has no necessary link to dispersion relations. As a result of this, the updated version is renamed to `rootfinder`. Similarly, after quadsections have been applied and the winding number is recalculated, method 0 calls for `WNContourPA` and method 1 calls for `WNChordContour`. Neither of these function names give any indication to what they compute. The parameter representing f'/f is given handle `fdf`, which is potentially ambiguous. Future versions give the handle `df_f` instead.

- The vector containing the roots, `r`, is output after each root is determined. This is suppressed in the updated version.

- The number of inputs to the majority of the functions is unnecessarily large. For example `dispersionroot` requires the user to input `dmyfun` (the derivative of f). Since the user has no need for f' the updated version will determine this value internally in `dispersionroot`. The function f'/f is only required internally to `WindNum`, but since it is determined in `dispersionroot` it must be passed between multiple functions as an input, such as in `Quadsec`. Further to this, `Quadsec` requires inputs `mffdf`, `myfun` and `dmyfun` which are the MATLAB function handle representation of `fdf`, the original function and its derivative respectively. Of these inputs, `Quadsec` ultimately only needs `myfun`. How this is achieved is outlined in Section 3.3.

- Although a minor problem, tabs have not been aligned with for loops in the program, nor have spaces around operators been used. This has been fixed, improving human readability.

- Finally, the help dialogues of each function do not clearly describe what the function does and in cases describe the inputs which were initially included and later removed.

3.3 Improved root finder

We will proceed by listing updates made to `dispersionroot`, detailing the improved file structure and concluding with a user guide.

3.3.1 Updates

A comprehensive list of changes made to `dispersionroot` will be given here including those which are relatively minor. This will be done as a chronological list. Some of the initial changes later become irrelevant as the file structure is changed but will be given for completeness. The function whose roots are to be determined will be denoted $f = f(z)$ throughout the list.

- Renamed `dispersionroot` to `rootfinder`.
- Suppressed vector containing roots, `r`, being output as it is updated during call to `rootfinder`.
- Removed need for f' as input to `rootfinder`, which is now calculated internally to `rootfinder`.
- Included optional input `'plot'` to `rootfinder`. This produces a plot of the roots. The title of the plot is given as the symbolic representation of `myfun`. Each root is given a unique colour by the MATLAB internal `hsv` function.
- Renamed variables and inputs to functions to remove ambiguity and improve readability.
- Specified `WindNum` to have a symbolic function input, `df_f` (representing f'/f), rather than a MATLAB function handle input, `matlabFunction(df_f)`. This is done since `WindNum` is the only function which calls `quadgk`, which is the only function requiring a MATLAB function handle input.

- Previously `WindNum` would call `quadgk` for each of the region's four edges. Now, `WindNum` calls `quadgk` once and uses the `'Waypoints'` input to `quadgk` to integrate over each edge, reducing computational expense.

- Allowed number of chords, `numchords`, to be an optional input to `rootfinder` required only if `method = 1` (the chord approach).

- Previously, if method 0 was chosen `WindNum` was called and if method 1 was chosen `ArgWN` was called by `rootfinder` to determine the winding number; these functions has now been combined into a single function, `methodchoice`.

- Previously, as described in Subsection 3.2.1, if a root is found near the boundary of the region before quadsectioning and if method 1 has been chosen then the program reverts back to method 0 after increasing the region size by 1% in every direction. The program now correctly uses method 1 to recalculate the winding number after such an instance.

- Some unnecessary inputs to `Quadsec` have been removed. Namely, the symbolic representation `df` of f', the symbolic representation `df_f` of f'/f, the MATLAB function handle `mfdf_f` of `df_f` and `maxquadsec` (which is a constant set to 20, defining the maximum number of times quadsectioning is allowed).

- The function `WindIntPoly` solves the weighted winding integral polynomials described at the end of Section 3.1. This is an ambiguous function name, and has consequently been changed to `polysolver`.

- In the original release, if the region was quadsectioned then `Quadsec` would call `WNContourPA` if method 0 was chosen or `WNChordContour` if method 1 was chosen to calculate the winding number of each subregion. However, `WindNum` (for method 0) and `ArgWN` (for method 1) already had the capability to determine the winding number for these subregions. Consequently `Quadsec` now calls

`methodchoice` (which is the combination of `WindNum` and `ArgWN`) to determine the winding number for subregions when using either method.

- Renamed `Quadsec` to `quadsec` for consistency of function naming.
- For method 1, `rootfinder` calls `methodchoice` to determine the winding number of a region; `ArgWNSide` is then called by `methodchoice` to determine the contribution to the winding number from each edge of the region by considering the change in argument of f. To remove ambiguity, `ArgWNSide` has been renamed to `argchange`.
- The original release made no use of tabs to align for loops nor spaces between operators. Tabs and spaces has now been included to improve human readability.
- Error checking for the inputs called with `rootfinder` have been implemented. This ensures, should the program crash, the user is told explicitly why.
- MATLAB style comments and help dialogues included for `rootfinder` and all of its subfunctions.
- In the original release, the primary output to `dispersionroot` was the winding number `WN` and the secondary output was the roots `r`. This order has been swapped.
- Allowed the winding number `WN` to be output only by choice.
- Prevented a MATLAB internal error being output when the winding number `WN = 0` or `0 < WN < 1`. When `WN = 0` a value was not being assigned to `r`, which has been corrected by assigning an empty vector `r = []`. For the case when `0 < WN < 1`, `r` is also assigned the empty vector and the user is warned of a non-integer winding number.
- Specified method 0 as the default method to `rootfinder` for reasons given in Section 3.2.1.

- Method 1 is chosen by including the optional input `'chords'` to `rootfinder`.
- The output vector containing roots now has entries ordered by increasing real part. This is primarily for use by `roottracker` which is discussed in Section `roottracker`.
- If the function specified for root finding is `f = 0` previous versions would output an error due MATLAB function `quadgk` integrating scalars. This is prevented by assigning `r = 0` at the beginning of `rootfinder` and returning the function. Although `rootfinder` is not intended to be used for trivial functions, this is implemented due to the development of `roottracker`. For a function $f = f(z;\tau)$, where we look to track roots as τ is incremented, occasionally a incremented value of τ results in $f = 0$ which requires the need for `rootfinder` to return `0` as a root.
- An optional parameter `'quiet'`, specified as the last input to `rootfinder`, has been implemented. This suppresses all script dialogue to the command line. This is particularly useful when calling `roottracker` which calls `rootfinder` multiple times.

3.3.2 Improved implementation in matlab

Firstly, all user inputs to `rootfinder` (formerly known as `dispersionroot`) are now checked to ensure they are correctly specified. A check determines if the correct number of inputs has been specified, accounting for variable inputs. The search region is checked to be a non-empty rectangular region in the complex plane.

If one variable input has been specified it is checked to be the string `'plot'` or `'quiet'`. If not, an error message is displayed. If two variable inputs have been specified the following cases satisfy the check, where inputs must be specified in order: {`'plot', 'quiet'`} or {`'chords', numchords`}. If three variable inputs have been specified the following cases satisfy the check, where inputs must be specified in order:

{'chords', numchords, 'plot'} or {'chords', numchords, 'quiet'}. Finally, if all four variable inputs have been specified they must be done in the following order: {'chords', numchords, 'plot', 'quiet'}. In each of these cases `numchords` is checked to be a positive integer.

The file structure of `rootfinder` has been simplified and shorted from the previous version. After inputs are checked, `methodchoice` is called to calculate the region's winding number `WN`.

```
[WN, err] = methodchoice(f, xsides, ysides, method, n);
```

By default the principle argument method is chosen, assigning the variable `method = 0`. If the optional `'chords'` method is chosen the variable `method` is assigned to value `1` and the optional parameter `numchords` is passed to the optional parameter `n` of `methodchoice`.

To determine the winding number by the chord approach `methodchoice` calls `argchange` for each of the boundary's edges. The parameter `err`, which is an optional output of `methodchoice`, notifies `rootfinder` a root is near a boundary if non-zero. If the winding number is non-integer then the search region is recursively increased by 1% in every direction until the winding number is integer.

For an integer winding number we proceed by three cases:

- If `WN = 0` then the vector containing the roots `r` is assigned to an empty vector.
- If `WN = 1, 2, 3` or `4` then `methodchoice` is called in a for loop running from `n = 1:WN` with `method = 2`. In the case of `method = 2` the optional parameter `n` of `methodchoice` is the weighting of the winding integral (3.3). That is, the integral of $z^n f'/f$ around the search region.
- If `WN > 4` then `quadsec` is called as follows:

```
r = quadsec(f, xsides, ysides, r, rn, quadnum, method,
   numchords);
```

with the majority of inputs as previous specified, as well as

- the number of roots `rn` in the entire region;
- the number of quadsections `quadnum` has recursively applied.

The region specified by `xsides` and `ysides` is split into four subregions by `quadsec`. The winding number for each subregion is then calculated by a call to `methodchoice`.

The winding number must be calculated around each subregion, which are specified by `xsides`, `ysides` and also the centre point of the quadsection `cent`. As a result, `quadgk` must be called with different parameters as used for determining the winding number of the region specified only by `xsides` and `ysides`.

To resolve this the optional input `n` is assigned to value `cent` and the modified call to `quadgk` is selected by assigning `method = 3` for the principle argument approach and `method = 4` for the chords approach.

A length four vector `WN` is outputted and contains the winding number of each subregion. Quadsectioning is then applied recursively until every subregion has a winding number of four or less, after which point the options for `WN <= 4` specified above are made in `rootfinder`, unlike in the previous version which duplicated this functionality in `quadsec`.

Finally the roots are ordered by increasing real part and, if `'plot'` is specified, a plot of the roots is made.

3.3.3 User guide

To run this program, MATLAB's Symbolic Math Toolbox is required. The top level control is entirely done through the function `rootfinder`. The additional scripts required to run this program (located in the same directory as `rootfinder`) are given in the following table, with complete listings in Code Listings.

methodchoice
argchange
quadsec
polysolver

Table 2: Additional scripts required to run `rootfinder`

We now outline the usage of `rootfinder`. The user specifies a function `f` (whose roots will be found) by `symfun`. The symbolic parameter must be exactly `z`, initialised by `syms`. The user also defines the real and imaginary limits of the complex region in which roots will be searched for, defined below as `xsides` and `ysides` respectively. This is the minimum number of parameters the user must specify and is exampled below.

```
syms z;
f = symfun(z + exp(z) + sin(z), z);
xsides = [-5, 5];
ysides = [-3, 3] * 1i;
r = rootfinder(f, xsides, ysides);
```

Up to four optional inputs are available to `rootfinder` and if used, must be specified in the following order: {`'chords'`, `numchords`, `'plot'`, `'quiet'`} after all compulsory inputs have been specified. These optional inputs will now be described.

By default, the principle argument method is chosen. Should the user wish to use chord approach the first optional input must be `'chords'` and the second optional input `numchords` (the number of chords each edge is to be split into to determine the winding number).

An optional plotting capability `'plot'` has been implemented and must be specified after `numchords` if the `'chords'` method has been chosen. The search region specified by `xsides` and `ysides` is plotted and each root is given a unique colour by the

MATLAB `hsv` function. The axes and title are automatically named and interpreted by LaTeX markup through use of MATLAB's `'interpreter'` property. An example figure produced by `'plot'` is below.

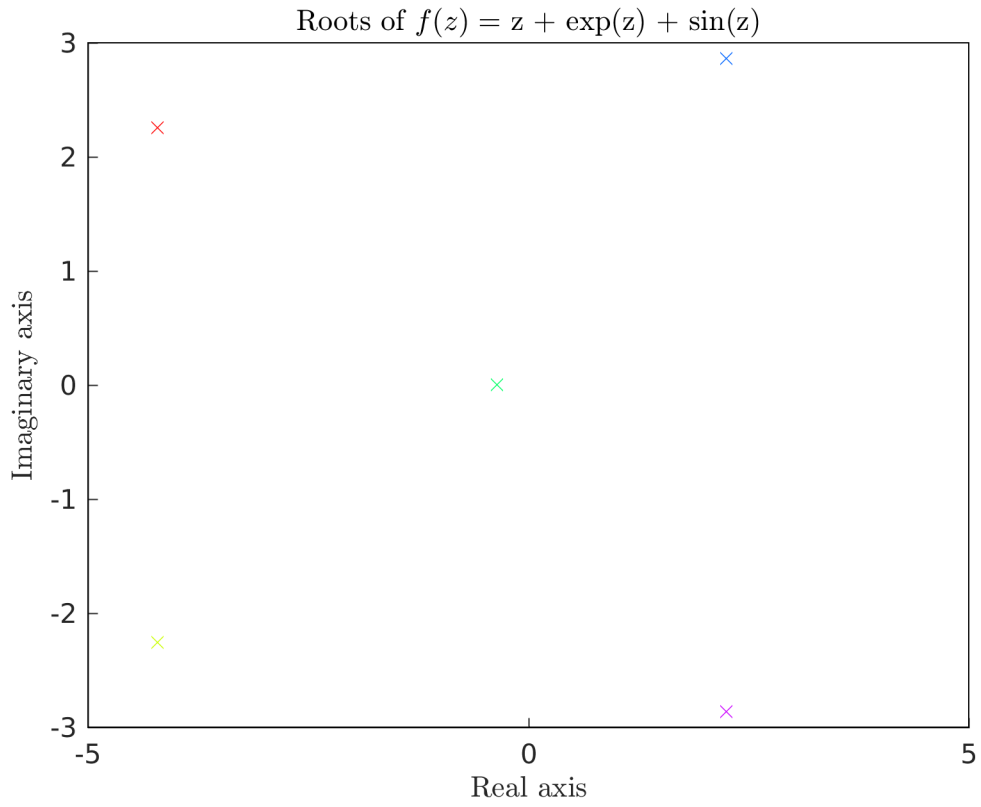


Figure 3.4: Example of `'plot'` plotting capability of `rootfinder`

Finally a user may suppress script dialogue to the command line outputted by `rootfinder` by specifying the optional input `'quiet'` as the final input.

4 Tracking Complex Roots

Consider a complex function $f = f(z;\tau)$ of a single argument and implicit parameter z and τ respectively. We consider the roots of f in a specified complex region for an initial value $\tau = \tau_0$ and aim to track these roots as τ is incremented.

4.1 Development and theory

Thales suggested a method of root tracking by a dynamic search region; a fixed size search region intended to move with the roots as they are incremented. This encompasses multiple complications, given in order of significance:

(i) computationally difficult to account for the case when a root leaves the search region as another root enters since the total number of roots is unchanged;

(ii) if one root diverges (from its initial value) more quickly than other roots then the search region would not be large enough to encompass all roots;

(iii) no indication of where an incremented root originates from;

(iv) Computationally expensive to search for roots in a large search region for a number of increments.

A possible solution to the second complication is to increase the search region's size if a root leaves it. This, however, places even more importance on the first complication.

We opted instead to track all roots found in an initial search region individually. We describe now the basic theory on how to achieve this before discussing its more complicated implementation into MATLAB.

Consider a function $f = f(z;\tau)$ of a single argument and implicit parameter z and τ respectively. It is reasonable to assume roots z of f are implicitly dependent on τ. We will therefore assume $z = z(\tau)$ is a root of f. If we increment τ by an infinitesimal

amount $\mathrm{d}\tau$, that is $\tau \to \tau + \mathrm{d}\tau$, then assuming roots are well-behaved our root z will also increase by an infinitesimal amount, $z \to z + \mathrm{d}z$. Note that we can consider $z = z(\tau)$ as a *path* of roots, parameterised by τ, and well-behaved roots corresponds to a well-behaved path.

Since $z = z(\tau)$ is a root of f, then

$$f(z(\tau); \tau) \equiv 0,$$

implying the full derivative of f vanishes, namely

$$\frac{\partial f}{\partial z}\frac{\mathrm{d}z}{\mathrm{d}\tau} + \frac{\partial f}{\partial \tau} = 0. \tag{4.1}$$

Let us now approximate the infinitesimals $\mathrm{d}\tau$ and $\mathrm{d}z$ by a known increment $\delta\tau$ and unknown increment δz respectively. Suppose z_0 is a root of f at $\tau = \tau_0$ which is known and z_1 denotes a root at $\tau = \tau_1 = \tau_0 + \delta\tau$ to be determined. We deduce that z_1 is dependent on z_0, τ_0 and $\delta\tau$. In particular,

$$z_1 = z_0 + \delta z \ \text{ for some unknown } \ \delta z = \delta z(z_0, \tau_0, \delta\tau).$$

Multiplying (4.1) by $\delta\tau$ and rearranging we find

$$\delta z \approx -\frac{\partial f}{\partial \tau}\,\delta\tau \left(\frac{\partial f}{\partial z}\right)^{-1}, \tag{4.2}$$

where derivatives are evaluated at (z_0, τ_0). Thus we can approximate the root z_j at $\tau = \tau_j = \tau_0 + j\delta\tau$ by recursively applying the above.

To exact this procedure, we use a large search region to locate initial roots at $\tau = \tau_0$. The implicit parameter is incremented ($\tau \to \tau_0 + \delta\tau$) and a linear approximation is determined for each root at this value of τ. A small search region is placed at each linear approximation to determine the exact location of the root. The process is repeated for the required number of increments. The following figure illustrates this methodology.

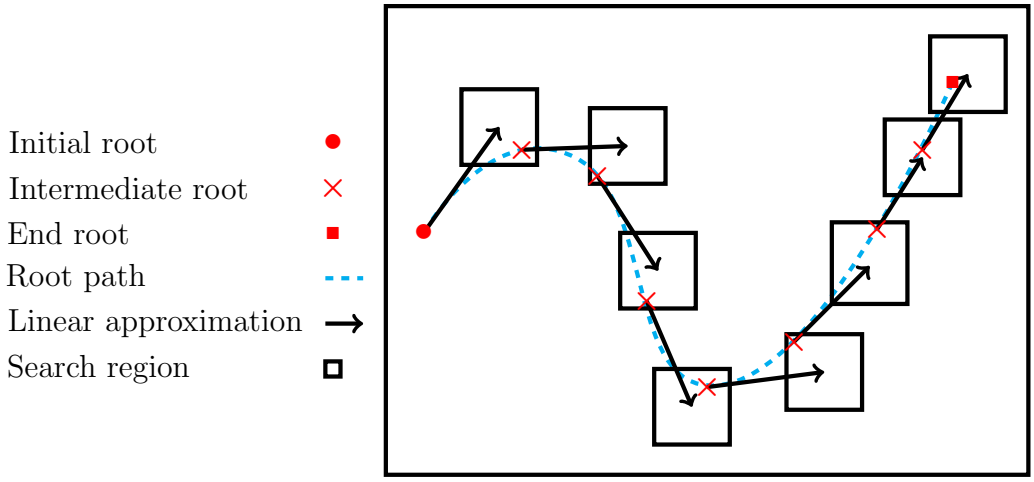
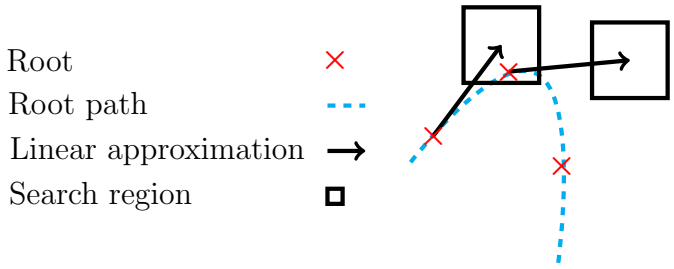
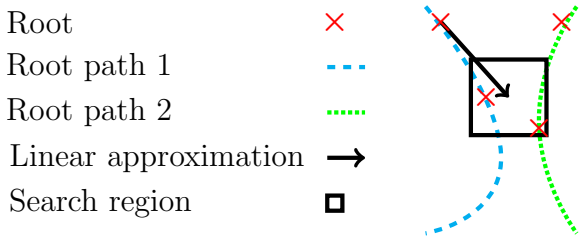


Figure 4.1: Illustration of root tracker

Two caveats come with such a method. Firstly, if the linear approximation is not accurate enough then no root will be found in the small search region. Secondly, if multiple roots are found in a small search region then there is no indication which root should be tracked. The following figures illustrate these complications.



(a) Illustration of no roots in small search region



(b) Illustration of multiple roots in a small search region

Figure 4.2: Complications in root tracking

How these complications are overcome will now be discussed in the implementation of the root tracking methodology in MATLAB.

4.2 Implementation in matlab

The function `roottracker` implements the theory described in Section 4.1. A symbolic handle `dz` is given to (4.2) which takes `z` and `tau` as variable inputs. We assign the initial value of the implicit parameter, `tau = tau_initial`. The function whose roots are to be tracked, `f(z, tau)`, has its dependency on `tau` removed by letting `f_tau(z) = f(z, tau)` allowing `rootfinder` to be called[4] using the initial search region specified as input to `roottracker`. We therefore have the initial roots of `f(z, tau)` (namely, at `tau = tau_initial`).

The approximate location of a root after incrementation is determined by inputting the initial root and the initial value of `tau` into `dz` and adding this to the initial root's value. A small square search region of default size `0.1` is placed at this approximation. This procedure is repeated for all roots.

We wish to now find once incremented roots of `f(z, tau)`. We update the implicit parameter, `tau = tau + dtau`, and the corresponding function `f_tau(z)`, whose roots are now the once incremented roots we seek; `rootfinder` is called to find the roots of `f_tau(z)` at each small square search region aforementioned.

We proceed by three cases.

- If a single root is found in a small search region it is assumed to be the correct once incremented root.

- If multiple roots are found in a small search region then the corresponding initial root will no longer be tracked (in particular, its value and all its subsequent incremented values will be set to `inf`). All remaining roots will continue to be tracked.

[4]Since `rootfinder` can only find roots functions with a single parameter.

- If no roots are found[5] in a small search region then the region is recursively increased in size by 5%. If a root is still not found after increasing the region size 6 times over (which corresponds to a compound increase of approximately 34%) then a sub-incrementation method is applied.

The sub-incrementation method is implemented by calling `roottracker` within `roottracker`. If it is the $(j+1)$th incremented root which cannot be determined then the program tracks from the jth incremented root with `sub_n_incr` increments of size `dtau / sub_n_incr` to find the $(j+1)$th incremented root. The variable `sub_n_incr` is defined globally to ensure its value is retained after `roottracker` is internally called. The global definition also allows the program to prevent sub-increments being used between sub-increments. Initially 10 sub-increments are used; if a root still cannot be found then 20 and 30 sub-increments are used before declaring the root cannot be tracked and setting its value and all subsequent increments to `inf`. Remaining roots continue to be tracked.

Initially, if required, only 2, 4 and 6 sub-increments were used. In practice it was found that when sub-increments were required a large number of them was usually needed due to a sharp turning point in a root's path. This is why the number of sub-increments was increased to 10, 20 and 30.

In the multiple roots case `roottracker` was initially designed to track all roots appearing in small search regions. This lead to complications in storing the new roots undergoing tracking and how to understand where the incremented root originates from. Another method to combat the multiple roots case was to apply the sub incrementation method although this was also met with complications. In practice, when roots meet in the same small search region (assuming it is suitably small) it often occurs as simple roots converge on a root of multiplicity greater than 1 for given a value of τ. For example, the function $f(z;\tau) = z^2 - \tau$ where $\tau_0 = -1$ is

[5]Before making improvements to the root finder this would make the program break with a MATLAB internal error since no value would be assigned to the vector containing the roots.

incremented 10 times by 0.1. There are two simple roots at $z = \pm\sqrt{\tau}$, however, on the tenth incrementation f has a double root at $z = 0$. To ensure this ambiguity is not overlooked it was opted for the program to no longer track roots in such a case; instead, the user is suggested to call `rootfinder` with the current dimensions of the small search region and the current value of τ, whose values will be output by `roottracker`in such a case.

The above is repeated for the specified number of increments to be added to τ. Throughout this process roots are stored in a matrix `r` which contains all initial roots and their incremented values.

Initially `roottracker` required user input when difficultly was met in tracking roots. The user would be asked if they wanted to increase the small search region size by up to 10%, 20% and 30% before asking if sub-increments should be used. In practice autonomy was far more productive as many increments are often required and it is impractical for the user to have a large amount of dialogue with the program. This also lead to the implementation of the `'quiet'` input to `roottracker`, which simply suppresses the often sizeable dialogue outputted by `roottracker` and by consequence `rootfinder`.

A plotting capability similar to that of `rootfinder` is implemented in `roottracker`. Two options are available by specifying the optional input `'plot'` or `'boxes'`. The option `'plot'` plots all roots and their incremented values whilst the option `'boxes'` additionally plots the initial search region and all small search regions. The distinction between between the two was made so that for users only interested in the root's path can have a clearer figure with `'plot'` and users interested to see if a root leaves an initial search region or in troubleshooting can use `'boxes'`. Different symbols have been chosen for end points of the roots' paths to ensure clarity.

4.3 Error checking

Similarly to `rootfinder`, all user inputs to `roottracker` are checked for validity. A check determines if the correct number of inputs to `roottracker` has been specified, the function `f` given is a symbolically defined function and the initial search region specified is a non-empty rectangular region in the complex plane. In addition, `tau_initial` and `dtau` are checked to be numerical complex and `n_incr` is checked to be a positive real integer.

If only one optional input is specified a check determines whether it is a positive real number, in which case this value is assigned to `root_box_size`. If non-numerical, the check determines if the input is a string identical to `'plot'` or `'boxes'`, in which case `plotting` is assigned to value `1` or `2` respectively (with a default value of `0` to indicate no plotting is required). If the input is a string identical to `'quiet'` the value `1` is assigned to a parameter `quiet`, suppressing all dialogue. If the optional input does not match these three cases then the program outputs an error with details on which input is invalid.

If two optional inputs have been specified then a check determines if the first input is a positive real number, in which case its value is assigned to `root_box_size` and that the second optional input is exactly the string `'plot'`, `'boxes'` or `'quiet'`, otherwise an error is outputted.

Finally if all three optional inputs are specified a check determines if the first input is a positive real number, in which case assigning the value to `root_box_size`. Following this, the second input is checked to be `'plot'` (or `'boxes'`) and the final input to be `'quiet'`, outputting an error otherwise.

Occasionally (4.2) will have a vanishing denominator. To account for this a simple check has been made. The `numden` function of the symbolic toolbox symbolically returns the denominator of its argument (whose inputs will be the previous root and increment value). If this is found to be vanishingly small then the small search region

will remain at the same location as the previous increment's small search region.

As described in Section 4.2, if an incremented root's search region contains no roots then the program increases the size of the small search region in increments of approximately 5%. If a root has still yet to be found after increasing the small search region by approximately 30% then `roottracker` applies a sub-incrementation method by internally calling itself. If it is the $(j+1)$th incremented root which cannot be determined, the program tracks from the jth incremented root with `sub_n_incr` increments of size `dtau / sub_n_incr` to find the $(j+1)$th incremented root. The variable `sub_n_incr` is defined globally to ensure its value is retained after `roottracker` is internally called. The global definition also allows the program to prevent sub-increments being used between sub-increments.

Initially `sub_n_incr = 10`. If any sub-incremented root values cannot be determined, small search regions are allowed to increase in size by up to approximately 30%. If a root still cannot be determine its value is set to `inf` and the call to `roottracker` breaks. The higher level call to `roottracker` repeats this process after increasing `sub_n_incr` to `20` and then again to `30` before declaring the root untrackable and assigning the root's value and all of its subsequent increments to `inf`. All other roots will continued to be tracked. After each time `roottracker` is interally called to use sub-increments the value of `sub_n_incr` is reset to `[]`. This ensures that if a different root requires sub-incrementation then the value of `sub_n_incr` only corresponds to that root.

If an incremented root's search region contains more than one root then the user is informed and the root's value and all of its subsequent increments are set to `inf`. The user is also informed of the value of `tau` and the geometry of the incremented root's search region, suggesting that `roottracker` be called with these as inputs. Should this warning be output for more than one root on the same increment it is likely multiple root paths which are being tracked meet nearby. Recommended recourse is to refine the root tracking, primarily by decreasing the magnitude of `dtau` but also by

decreasing `root_box_size`.

4.4 User guide

To run this program, MATLAB's Symbolic Math Toolbox is required. The top level control is done entirely through the function `roottracker`. The additional scripts required to run this program (located in the same directory as `roottracker`) are given in the following table, with complete listings in Code Listings.

rootfinder
methodchoice
quadsec
polysolver

Table 3: Additional scripts required to run `roottracker`

We now outline the usage of `roottracker`. The user specifies a function `f` (whose roots will be tracked) by `symfun`. This function must have exactly `z` and `tau` as its complex symbolic parameters, initialised by `syms`, where `z` and `tau` are the single argument and implicit parameter of `f` respectively. The initial value of `tau` must be specified, written in the following listing as `tau_initial`. Roots for this value of `tau` are found inside a complex region whose real limits are specified by `xsides` and complex limits `ysides`. Roots are then tracked as `tau` is incremented by a user specified amount, `dtau`, which can take complex values. The user also chooses how many increments, `n_incr`, of `dtau` are to be added to `tau`.

This is the minimum number of parameters the user must specify and is exampled below.

```
1 syms z tau;
2 f = symfun(2 * tau * z * cos(2 * z) - sin(2 * z) * (tau^2 - z
    ^2), [z tau]);
```

```
3
4  xsides = [-5, 5];
5  ysides = [-1, 1] * 1i;
6
7  dtau = 0.05 + 0.057143i;
8  tau_initial = dtau;
9  n_incr = 45;
10
11 r = roottracker(f, xsides, ysides, tau_initial, dtau, n_incr);
```

The output `r` contains all roots originating from the initial conditions as `tau` is incremented[6]. The initial roots are ordered by increasing real part and this order is kept throughout incrementation. The (`i + 1`)th row of `r` contains the roots of `f` after `i` increments have been added to `tau_initial` (meaning the first row contains the initial roots) and the `j`th column of `r` contains all increments of the `j`th root of `f`.

Up to three optional inputs are available to `roottracker` and if used, must be specified in the following order: {`root_box_size, 'plot', 'quiet'`} or {`root_box_size, 'boxes', 'quiet'`}.

By default the small search regions placed at the centre of linear approximations are square with width `root_box_size = 0.1`. If the program outputs a warning message saying that multiple roots were found in a single incremented search region then decreasing the value of `root_box_size` may solve this problem. This can be combined with decreasing `dtau` and increasing `n_incr` accordingly.

An optional plotting capability `'plot'` similar to that of `rootfinder` (Section 3.3.3) has been implemented. A unique colour is assigned to each root and all of its increments by the MATLAB `hsv` function. Initial roots are marked with black filled circles and final incremented roots are marked with black filled squares. Intermediate roots are marked with crosses. The axes and title are automatically named and

[6] Although a tracked root may leave the region specified by `xsides` and `ysides`.

interpreted by LaTeX markup through use of MATLAB's `'interpreter'` property. Specifically, the title is given as the function name in question followed by the endpoints of `tau`. An example figure produced by `'plot'` is below.

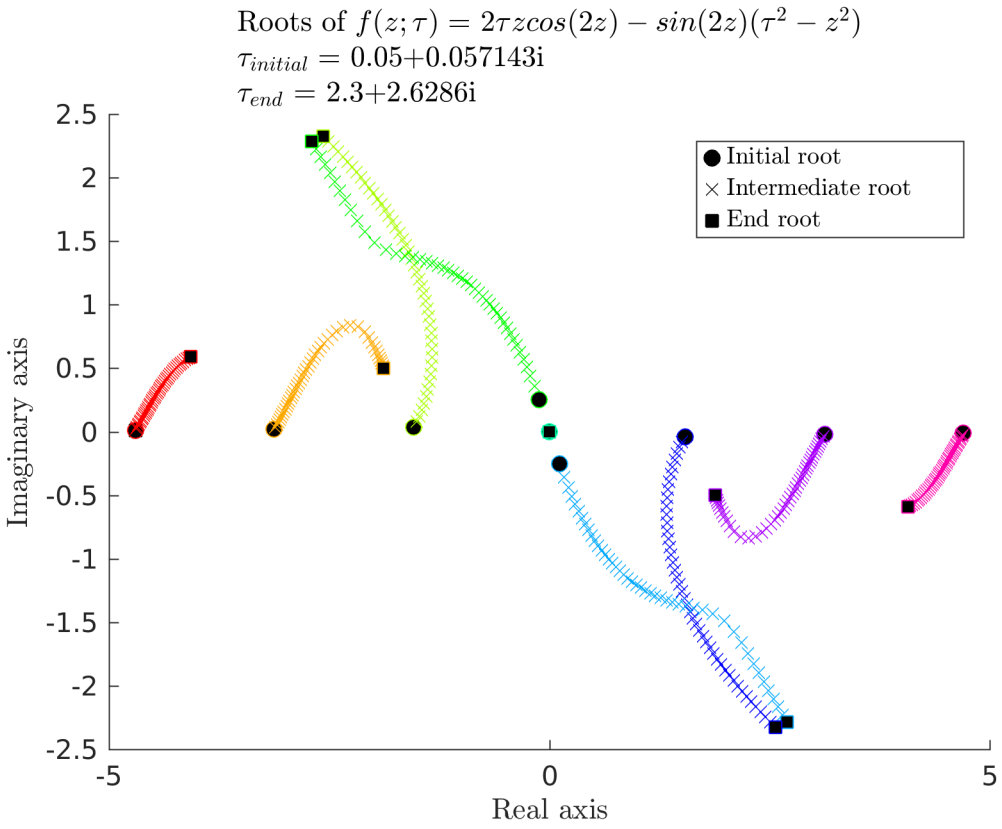


Figure 4.3: Example of `'plot'` plotting capability of `roottracker`

The search regions for each root can also be plotted over each root, done by replacing `'plot'` with `'boxes'`. This includes the initial search region specified by `xsides` and `ysides`. This option is recommended to be used if multiple roots are being found in a single box, to check if roots leave the initial search region and for other such validation checking done by the user. To illustrate this, the following figure is produced by calling the same inputs to `roottracker` as in Figure 4.3 and also specifying the optional input `root_box_size = 0.4`.

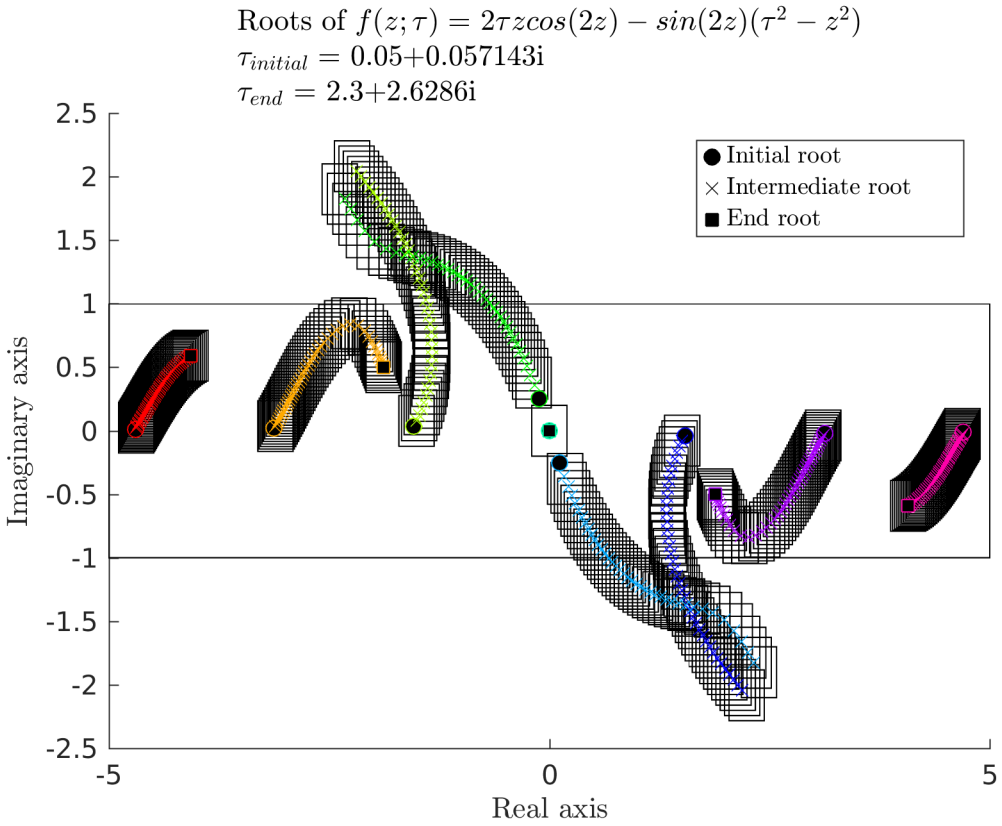


Figure 4.4: Example of `'boxes'` plotting capability of `roottracker`

Naturally a larger `root_box_size` allows for fewer increments to be used, however, this sacrifices the accuracy achieved from a smaller valued `root_box_size`. We note that due to the increase in `root_box_size` the roots which leave the initial search region shown in Figure 4.4 are no longer tracked as their paths come in close proximity. This is because multiple roots are found in a single small search region; this behaviour is not tracked by `roottracker` with reason given in Section 4.1.

Typical dialogue produced by `roottracker` includes the number of roots being tracked and the number of increments currently added. Specifying the optional parameter `'quiet'` initialises a parameter `quiet` and sets its value to `1`. This suppresses all script dialogue produced by `roottracker` (and by consequence `rootfinder`).

Example 4.1. Suppose we wish to determine the roots of

$$f(z) = J_0(zi)i - z\frac{\partial}{\partial z}J_0(zi),$$

where J_0 is the Bessel function of the first kind, order 0. It is known that for real arguments, Bessel functions of the first kind and their derivatives (which can be defined by higher order Bessel functions of the first kind) have real roots. Using this knowledge we will track roots of

$$f(z;\tau) = J_0(\tau z)\tau - z\frac{\partial}{\partial z}J_0(z\tau),$$

as τ is incremented from unit real to unit imaginary. This is achieved by the following listing, where `besselj(NU, z)` is the MATLAB command defining the Bessel function of first kind, order `NU` and argument `z`:

```
syms z tau;
f(z, tau) = symfun(z * diff(besselj(0, tau * z), z) + tau *
   besselj(0, tau * z),z);

xsides = [0 10];
ysides = [-0.1 0.1] * 1i;

tau_initial = 1;
dtau = -0.05 + 0.05i;
n_incr = 20;

r = roottracker(f, xsides, ysides, tau_initial, dtau, n_incr,
   'boxes', 'quiet');
```

producing the following plot.

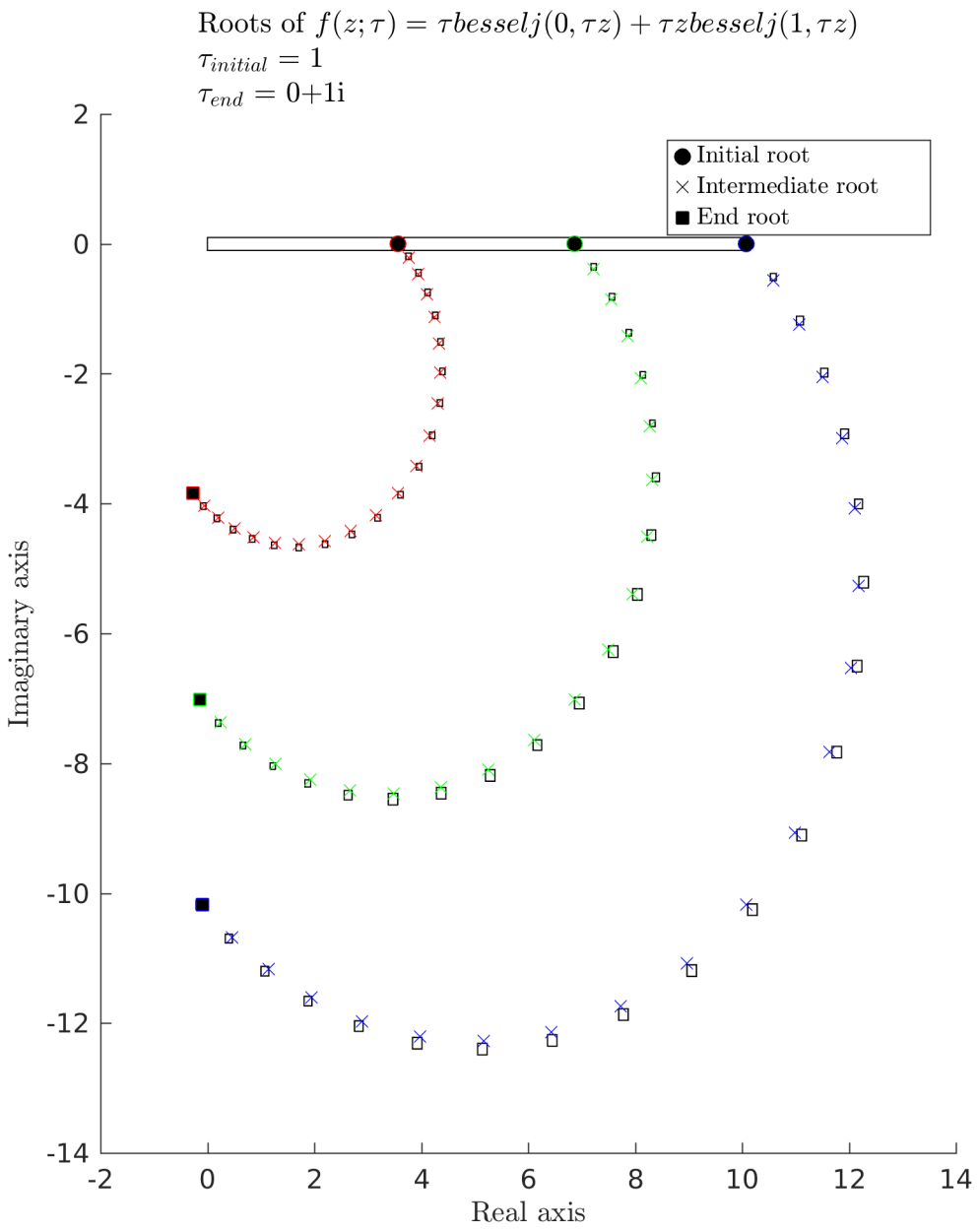


Figure 4.5: Example of root tracking

Note that only the roots which were found for the initial value of τ are tracked, meaning there may be other roots in the region displayed in Figure (4.5).

5 Application to Representative Problem

In this section we will consider a truncated approximation of the modal solution derived in Section 2. Throughout this section, the solution is computed up to a height $z = 10$. We will consider two specific test cases.

In both cases, roots of the governing dispersion relation

$$\mu \sin \mu + S \cos \mu = 0, \tag{5.1}$$

derived in Section 2 will be found with and without root tracking. The truncated modal solutions established from these methods will be compared against the boundary condition at the base of the duct before being compared with one another.

5.1 Case 1: without singularities

We pay particular focus to $\kappa = 3$, $S = 4 + 2.3i$ and $f(x) = S(x^2 - 1) + 2$, where such a form for f has been chosen to avoid singularities in the corners of the duct; this choice of f is symmetric in the range $x \in [-1, 1]$ as requisite in Section 2.

5.1.1 Modal solution without root tracking

We search for roots of the dispersion relation (5.1) without tracking within a search region with real limits $[0, l]$ and imaginary limits $[-i, i]$. We consider $l = 25$ and $l = 50$, for which we have and 8 and 15 modes respectively. The approximations for these values of l are given in the following figure.

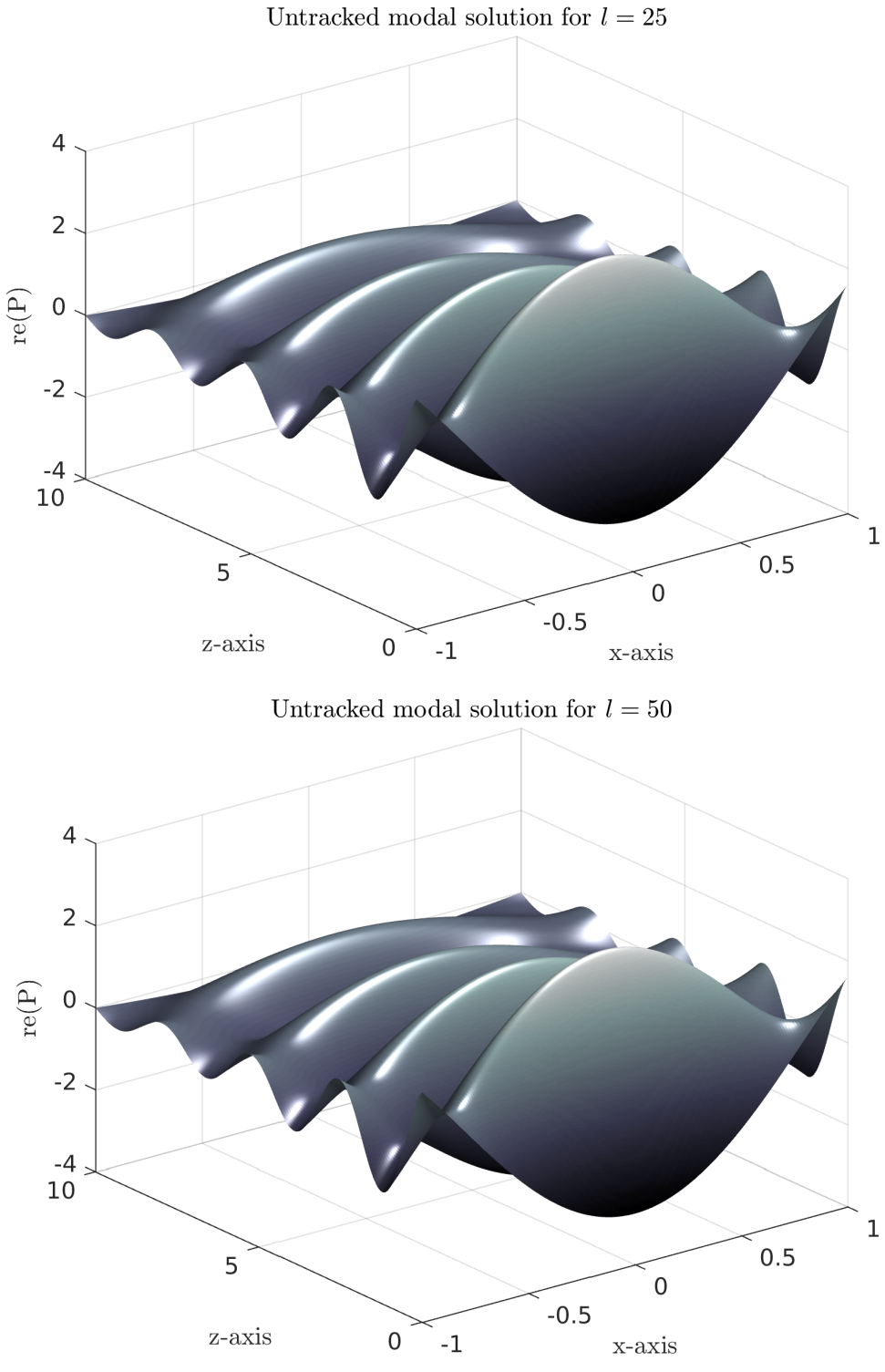


Figure 5.1: Modal solutions comparison for untracked roots within limits $[0, l], [-i, i]$

The two surface plots are graphically indistinguishable. The following figure is a surface plot of the difference between truncated modal solutions for $l = 25$ and $l = 50$.

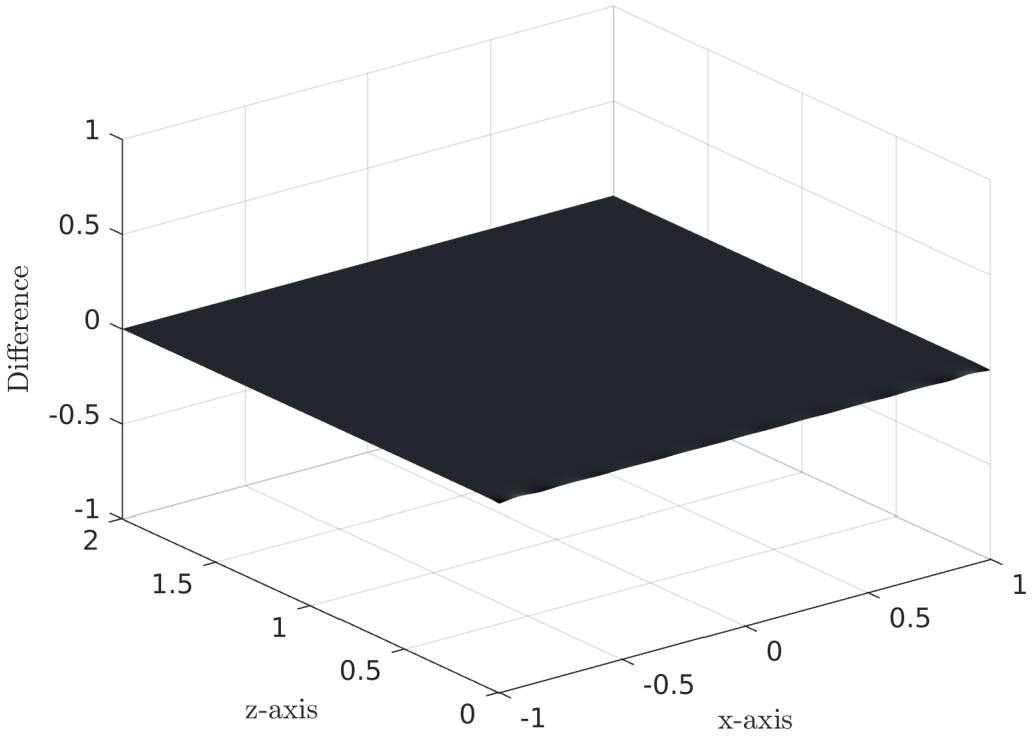


Figure 5.2: Difference of untracked modal solutions for $l = 25$ and $l = 50$

Figure 5.2 shows a flat plane, indicating little difference between truncated modal solutions for $l = 25$ and $l = 50$. In support of this, the maximum difference between the two solutions is of order 10^{-5}. Therefore the truncated modal solution has sufficiently converged for $l = 50$. The roots found for $l = 50$ are given in the following figure, produced by the `'plot'` plotting capability of `rootfinder`.

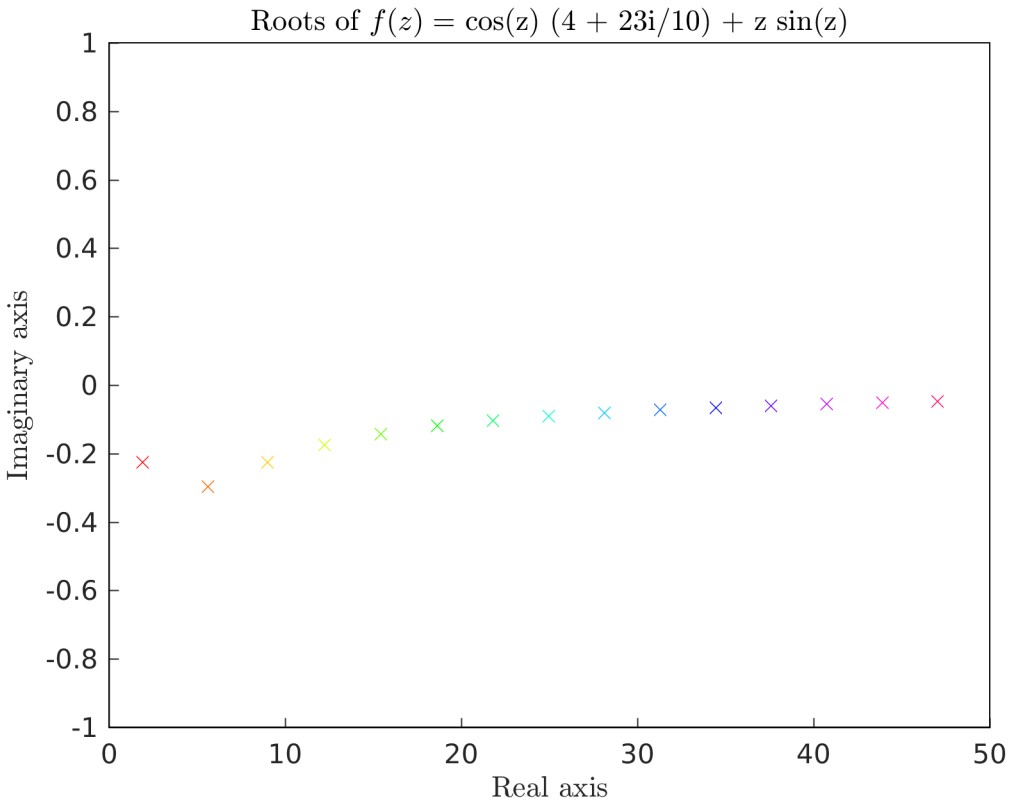


Figure 5.3: Roots found without tracking

The following figure compares the exact boundary condition (2.10c) at the base of the duct with the truncated modal solution obtained without root tracking.

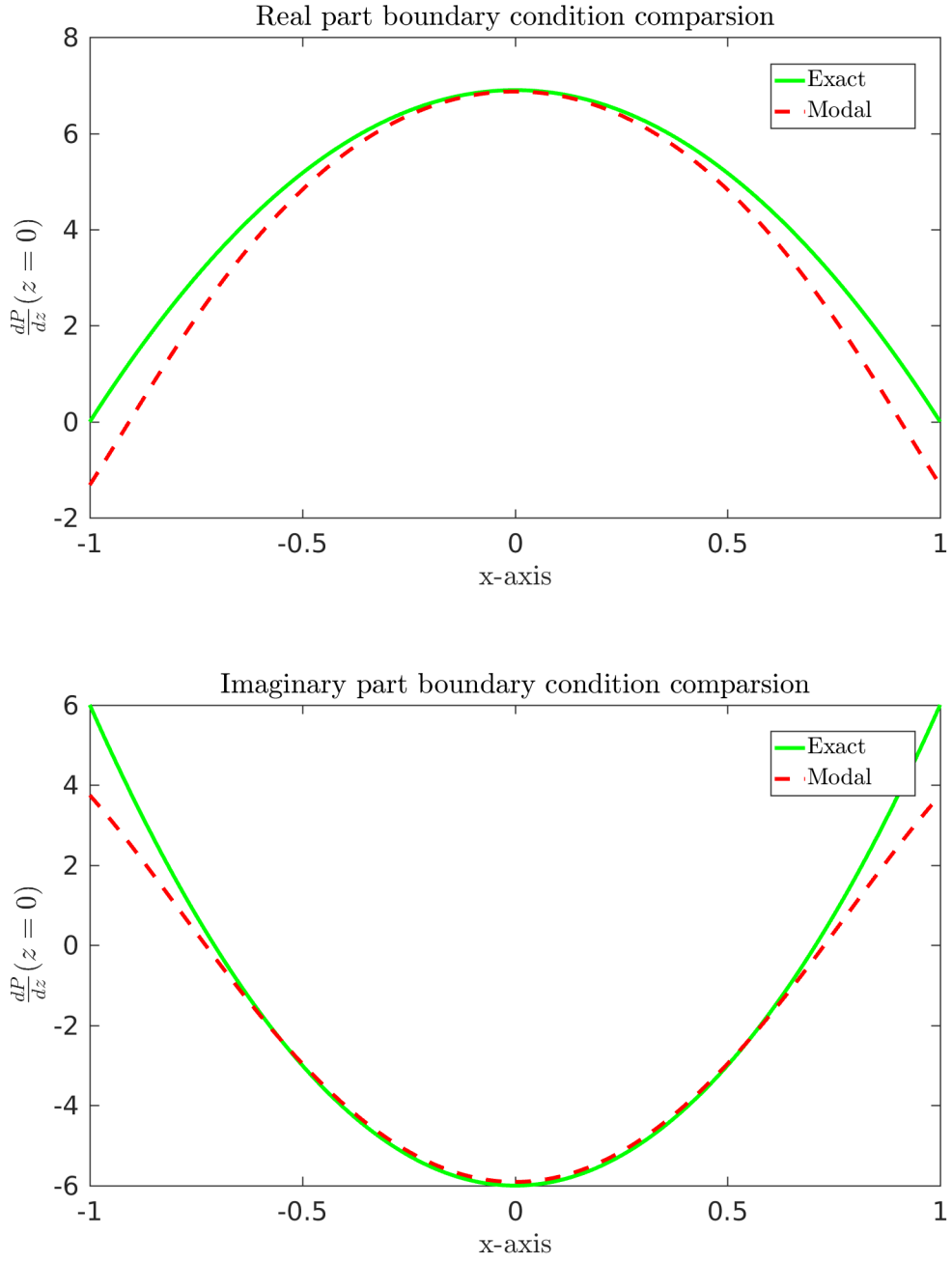


Figure 5.4: Boundary condition comparison for modal solution without tracking

Clearly the boundary condition is not satisfied, however, the approximation has

converged. Evidently a number of significant modes are missing from from the approximation. We will apply root tracking to ensure all roots of (2.15) are found.

5.1.2 Modal solution with root tracking

For the hard-walled case of $S = 0$, BVP (2.10) is self-adjoint. In this case the symmetric dispersion relation (2.15) is

$$\mu \sin \mu = 0, \tag{5.2}$$

whose roots are accordingly real and of the form $\mu_n = n\pi$ for n non-negative (by theory of superposition). Consequently we aim to track roots of (2.10) from $S = 0$ to $S = 4 + 2.3i$. Problematically, equation (5.2) has a double root at $S = 0$ which splits into two simple roots for $S \neq 0$. As discussed in Section 4.1, the root tracker has been chosen not to track roots which exhibit such behaviour. To avoid this behaviour the initial value of S will be δS, the small amount by which S is to be incremented. The path chosen to increment S over is defined by holding the argument of S constant. For illustrative purposes the initial search region will again be chosen to have real limits $[0, l]$ and imaginary limits $[-i, i]$. In practice vanishing values of δS allow vanishing imaginary limits.

We consider $l = 25$ and $l = 50$, for which we have 9 and 16 modes respectively. The approximations for these values of l are given in the following figure.

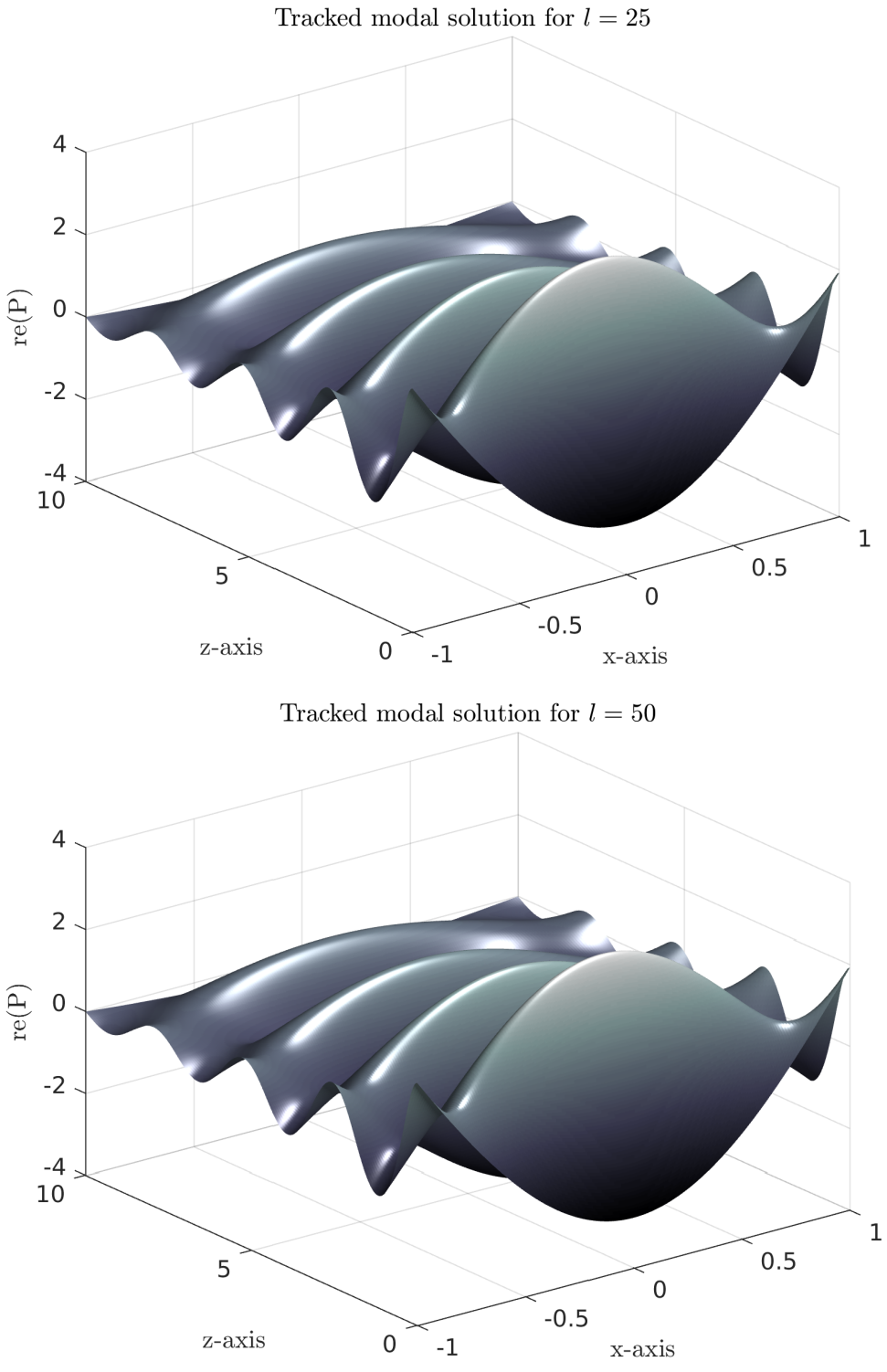


Figure 5.5: Modal solutions comparison for tracked roots within limits $[0, l], [-i, i]$

Again, the two surface plots are graphically indistinguishable. The following figure is a surface plot of the difference between truncated modal solutions for $l = 25$ and $l = 50$.

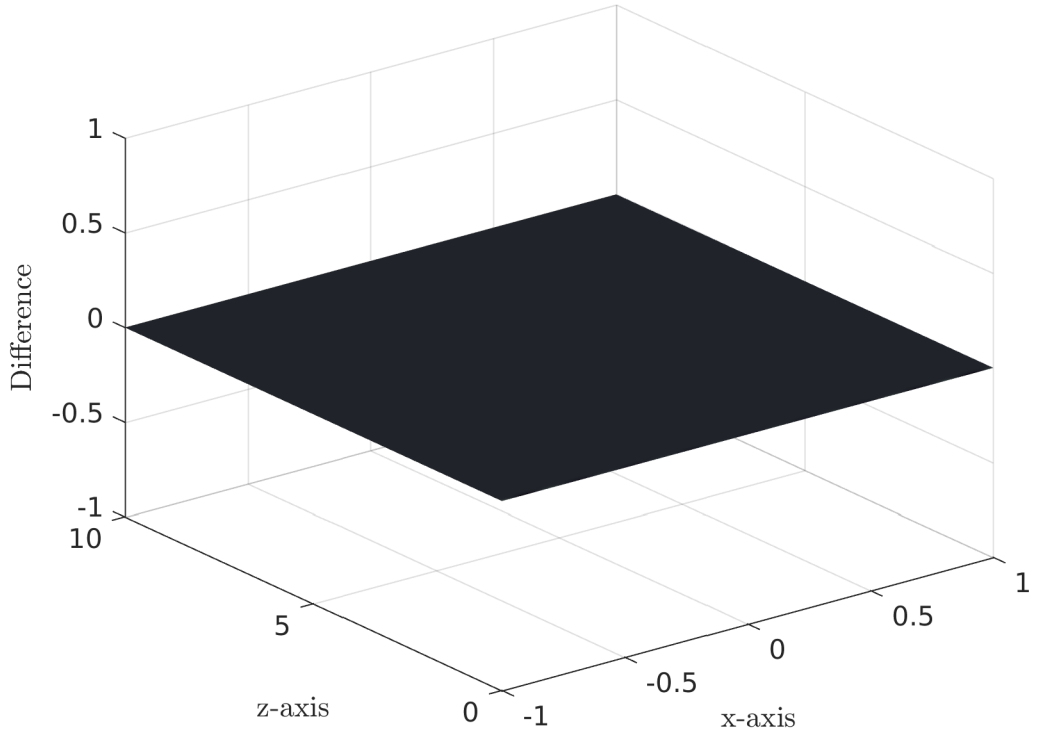


Figure 5.6: Difference of tracked modal solutions for $l = 25$ and $l = 50$

Similarly to Figure 5.2, Figure 5.6 shows a flat plane indicating little difference between truncated modal solutions for $l = 25$ and $l = 50$. In support of this, the maximum difference between the two solutions is of order 10^{-5}. Therefore the truncated modal solution has sufficiently converged for $l = 50$. The roots found for $l = 50$ are given in the following figure, produced by the `'boxes'` plotting capability of `roottracker`.

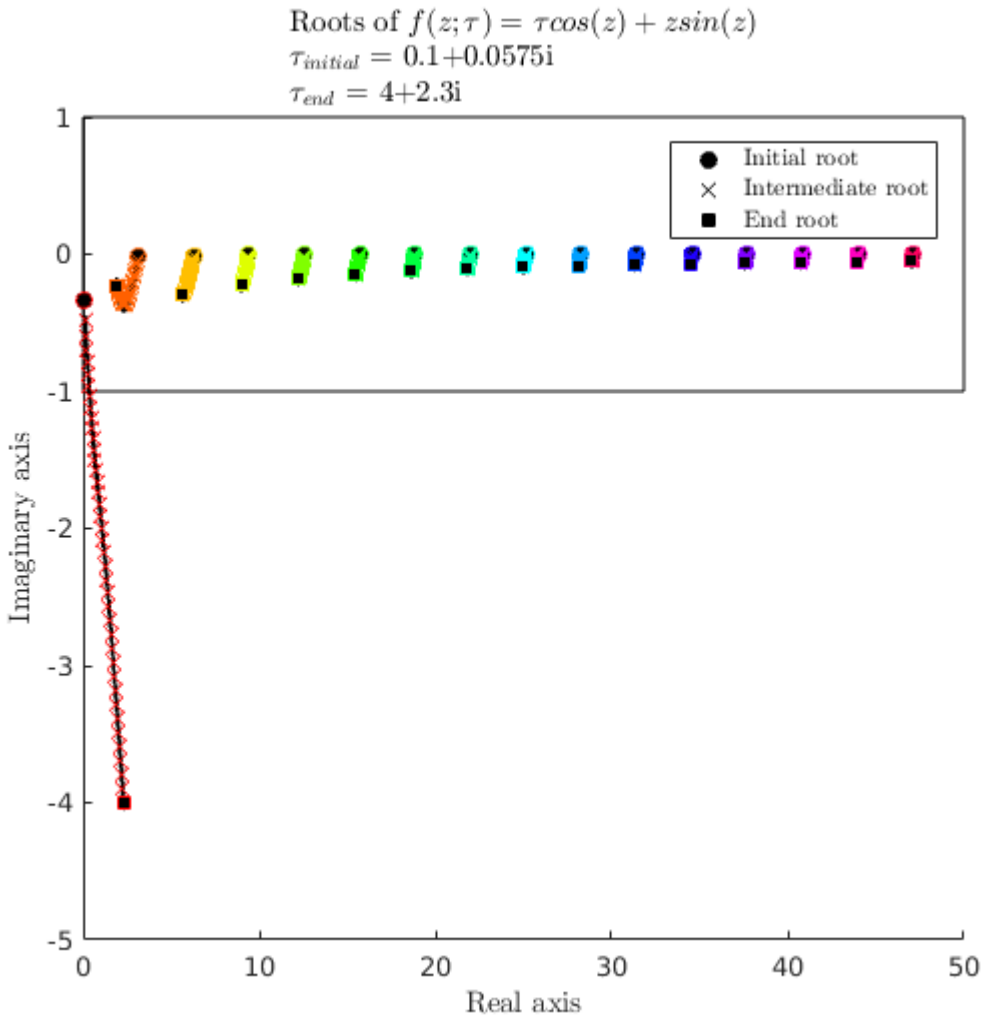


Figure 5.7: Roots found with tracking

Figure 5.7 shows a root leaves the initial search region, indicating the mode corresponding to this root is missing from the truncated modal solution without root tracking in Section 5.1.2. Let us again compare the exact boundary condition (2.10c) with the approximate modal solution obtained with root tracking.

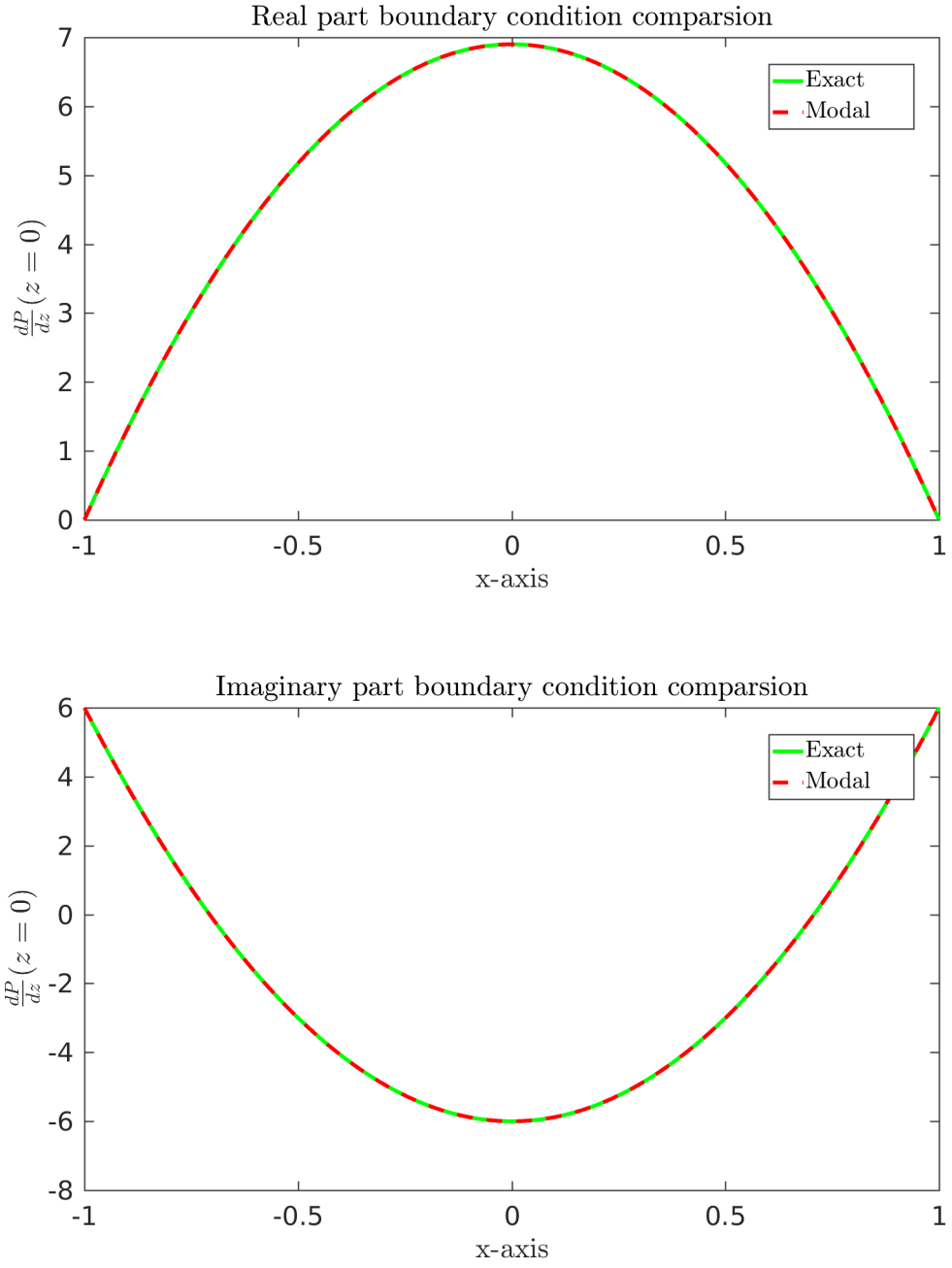


Figure 5.8: Boundary condition comparison for modal solution with tracking

The boundary condition has been matched with near perfect accuracy, allowing us

to confidently assume the full solution is accurate.

The following figure is a surface plot of the difference between approximate modal solutions obtained with and without root tracking. The z axis has been restricted to the range $[0, 2]$ to emphasise the difference in solutions. This is since a truncated modal solution with only a single mode will converge to 0, meaning a large range of z will not give a good indication to the difference in solutions.

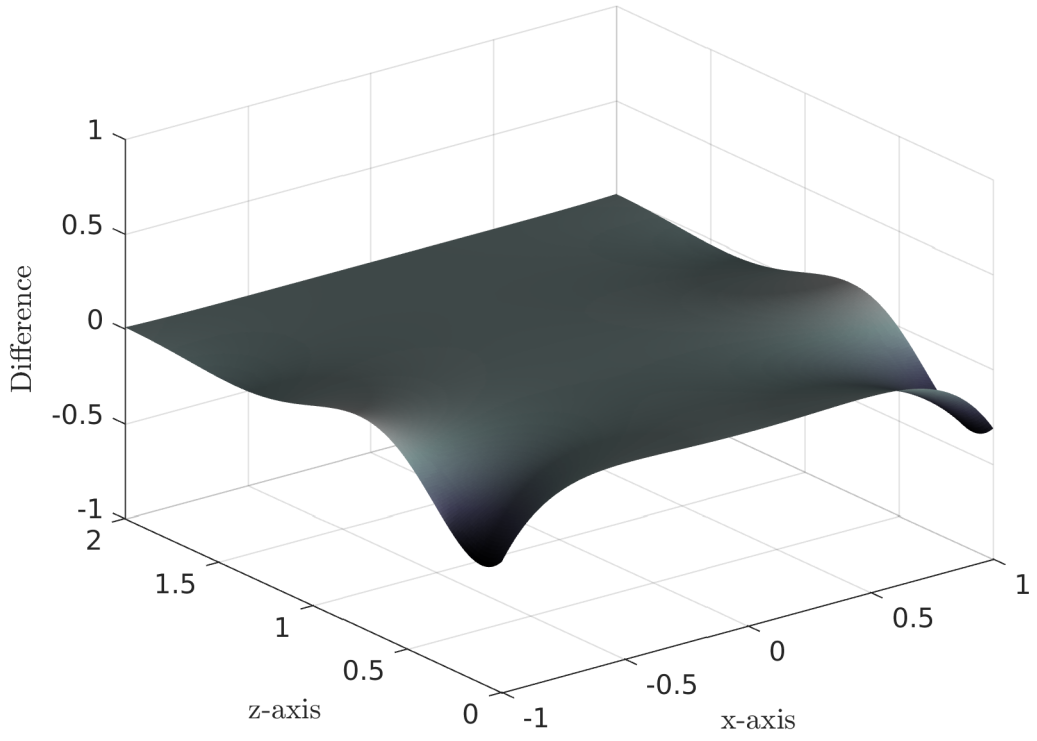


Figure 5.9: Difference of truncated modal solutions with and without root tracking

Figure 5.9 shows inconsistency between truncated modal solutions with and without root tracking, indicating the benefit of root tracking. We note that the truncated modal solutions with root tracking could be determined *without* root tracking by increasing the height of the search region, however, this observation is made under a posteriori knowledge.

5.2 Case 2: with singularities

We now consider a more dramatic case. Consider $\kappa = 3$, $S = 4+5i$ and $f(x) = 1$. This choice of f is symmetric in the range $x \in [-1, 1]$ as requisite in Section 2. Singularities form in the corners of the duct because of this form of f. Many more modes are required for a convergent truncated modal solution than in Section 5.1 due to these singularities.

The analysis of approximate modal solutions made in Section 5.1 will now be repeated for these parameters.

5.2.1 Modal solution without root tracking

We search for roots of the dispersion relation (5.1) without tracking within a search region with real limits $[0, l]$ and imaginary limits $[-i, i]$. We consider $l = 150$ and $l = 250$, for which we have 47 and 79 modes respectively. The approximations for these values of l are given in the following figure.

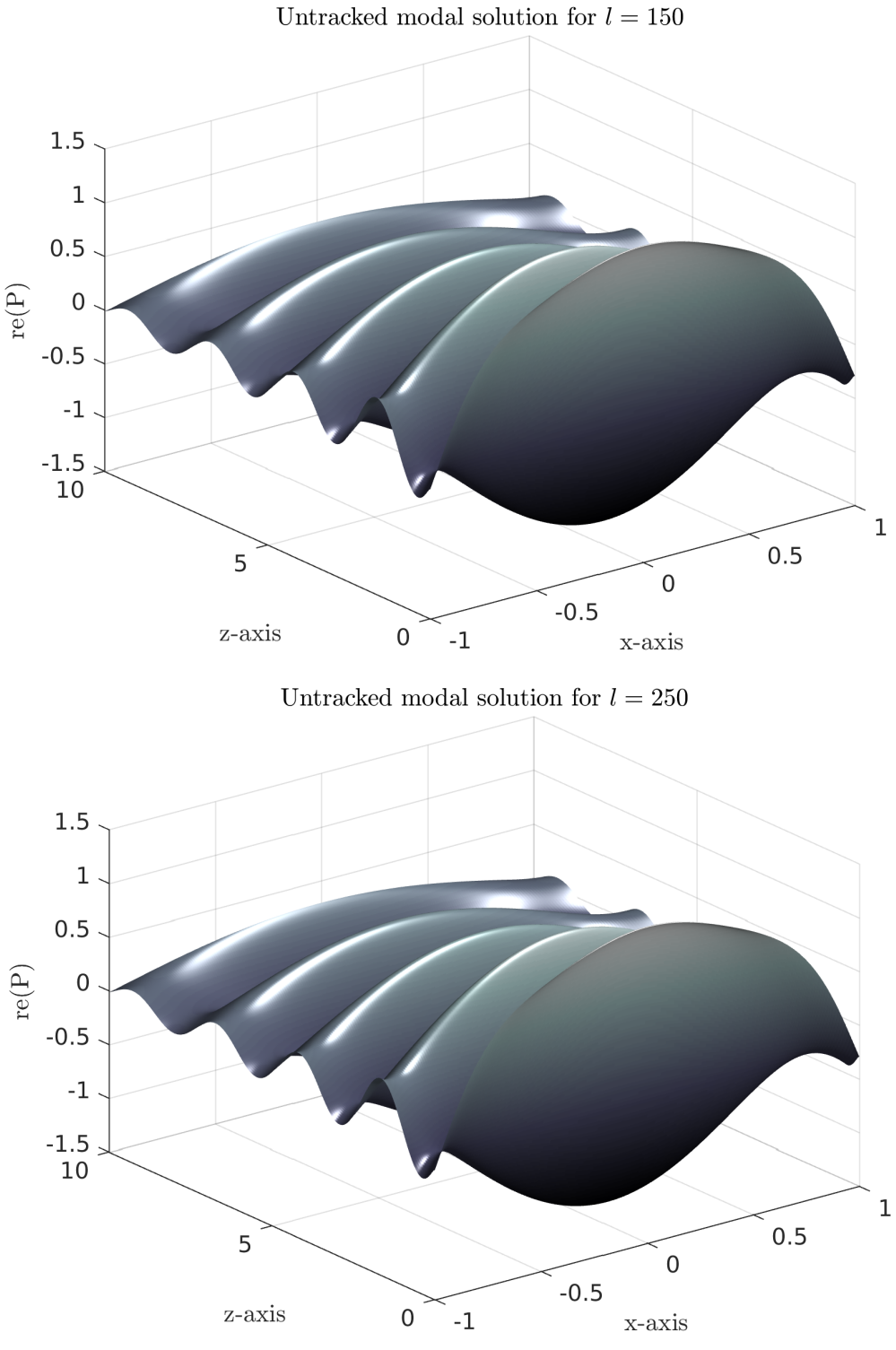


Figure 5.10: Modal solutions comparison for untracked roots within limits $[0, l], [-i, i]$

The two surface plots are graphically indistinguishable. The following figure is a surface plot of the difference between truncated modal solutions for $l = 150$ and $l = 250$.

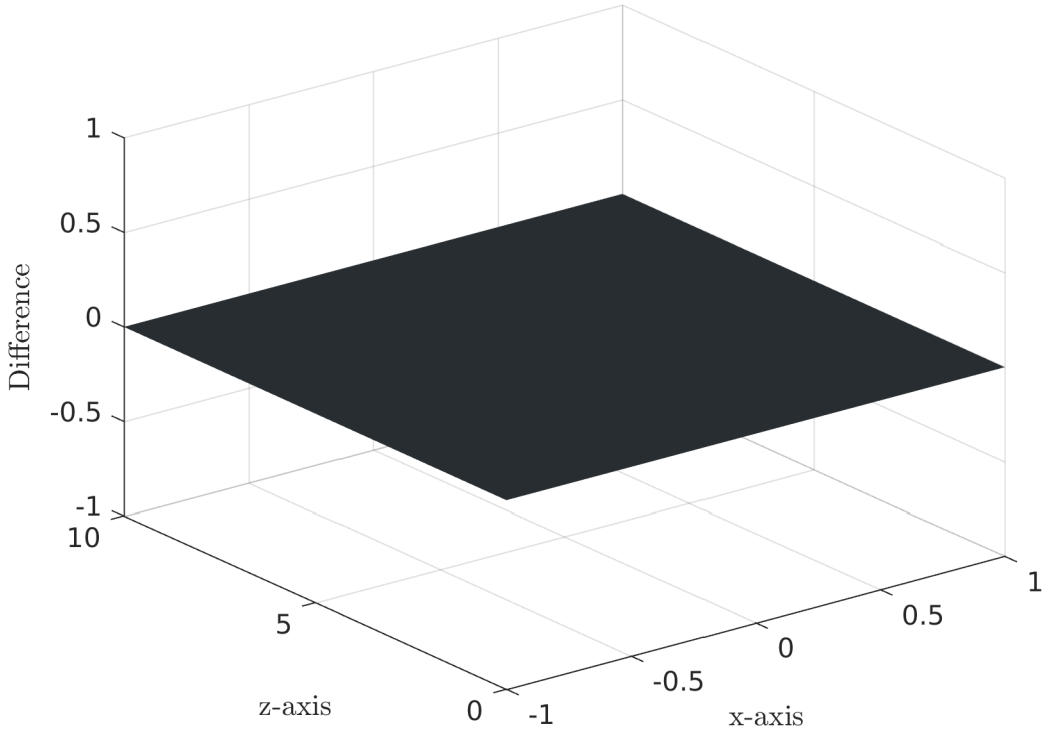


Figure 5.11: Difference of untracked modal solutions for $l = 150$ and $l = 250$

Figure 5.11 shows a flat plane, indicating little difference between truncated modal solutions for $l = 150$ and $l = 250$. In support of this, the maximum difference between the two solutions is of order 10^{-5}. Therefore the truncated modal solution has sufficiently converged for $l = 250$. The roots found for $l = 250$ are given in the following figure, produced by the `'plot'` plotting capability of `rootfinder`.

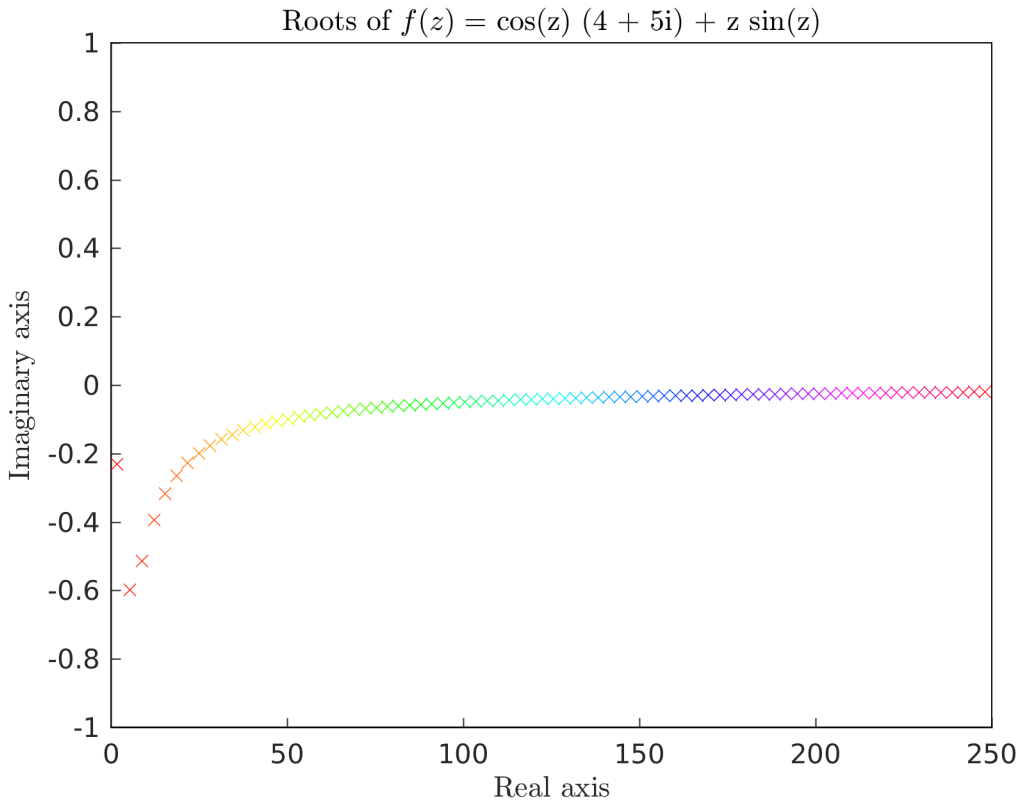


Figure 5.12: Roots found without tracking

The following figure compares the exact boundary condition (2.10c) at the base of the duct with the truncated modal solution obtained without root tracking.

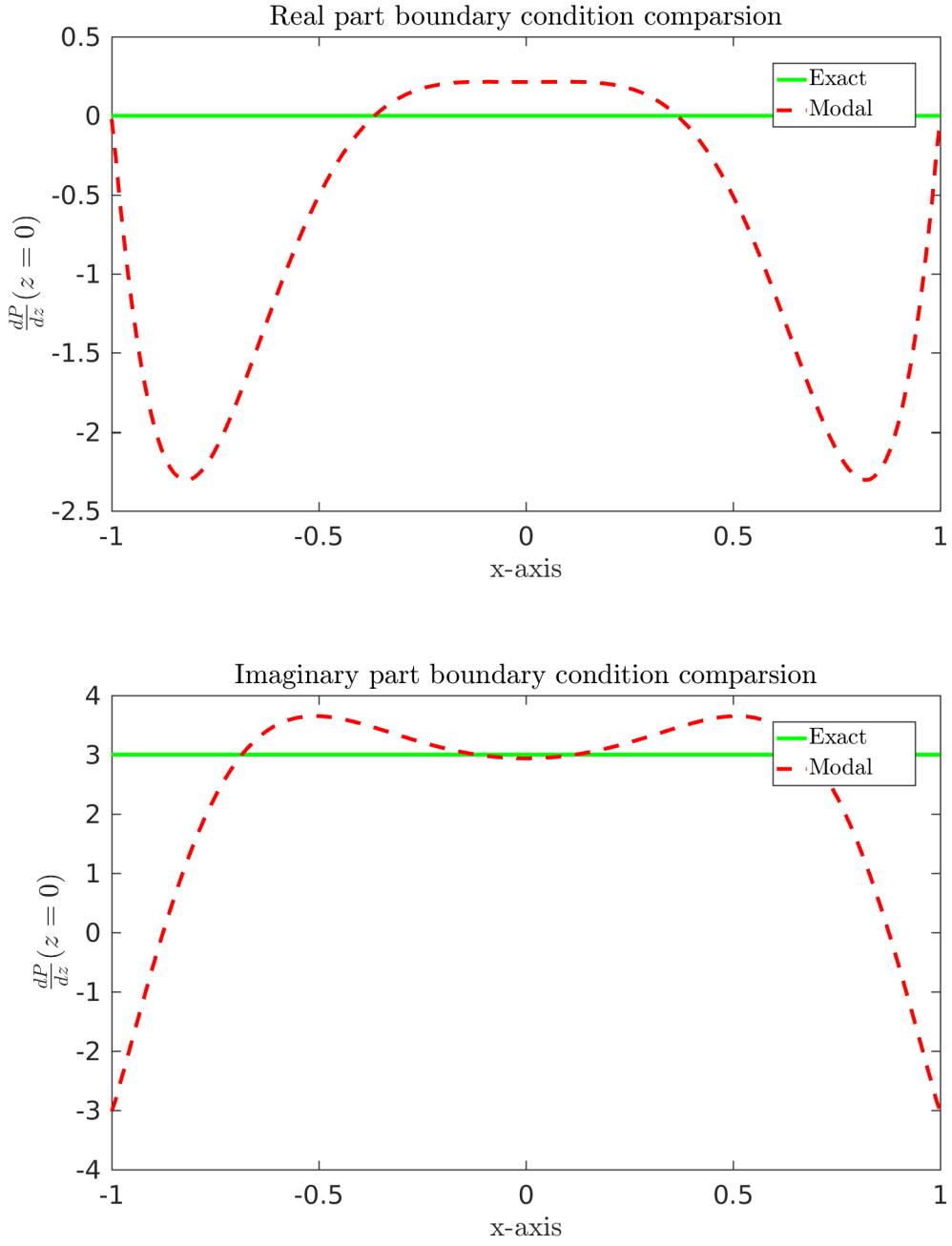


Figure 5.13: Boundary condition comparison for modal solution without tracking

Figure 5.13 shows the truncated modal solution without root tracking provides

a wholly inadequate approximation. Evidently a number of significant modes are missing from from the approximation. We will apply root tracking to ensure all roots of (2.15) are found.

5.2.2 Modal solution with root tracking

As in Section 5.1.2, we first consider the hard-walled case of $S = 0$ for which BVP (2.10) is self-adjoint. We track roots by the same methodology, choosing the initial value of S to be δS, the value by which S is incremented. For illustrative purposes the initial search region will again be chosen to have real limits $[0, l]$ and imaginary limits $[-i, i]$.

Due to the singularities in the corners of the duct infinitely many modes are required for a convergent modal solution. We consider $l = 1000$ and $l = 1250$, for which we have 319 and 398 modes respectively. The approximations for these values of l are given in the following figure.

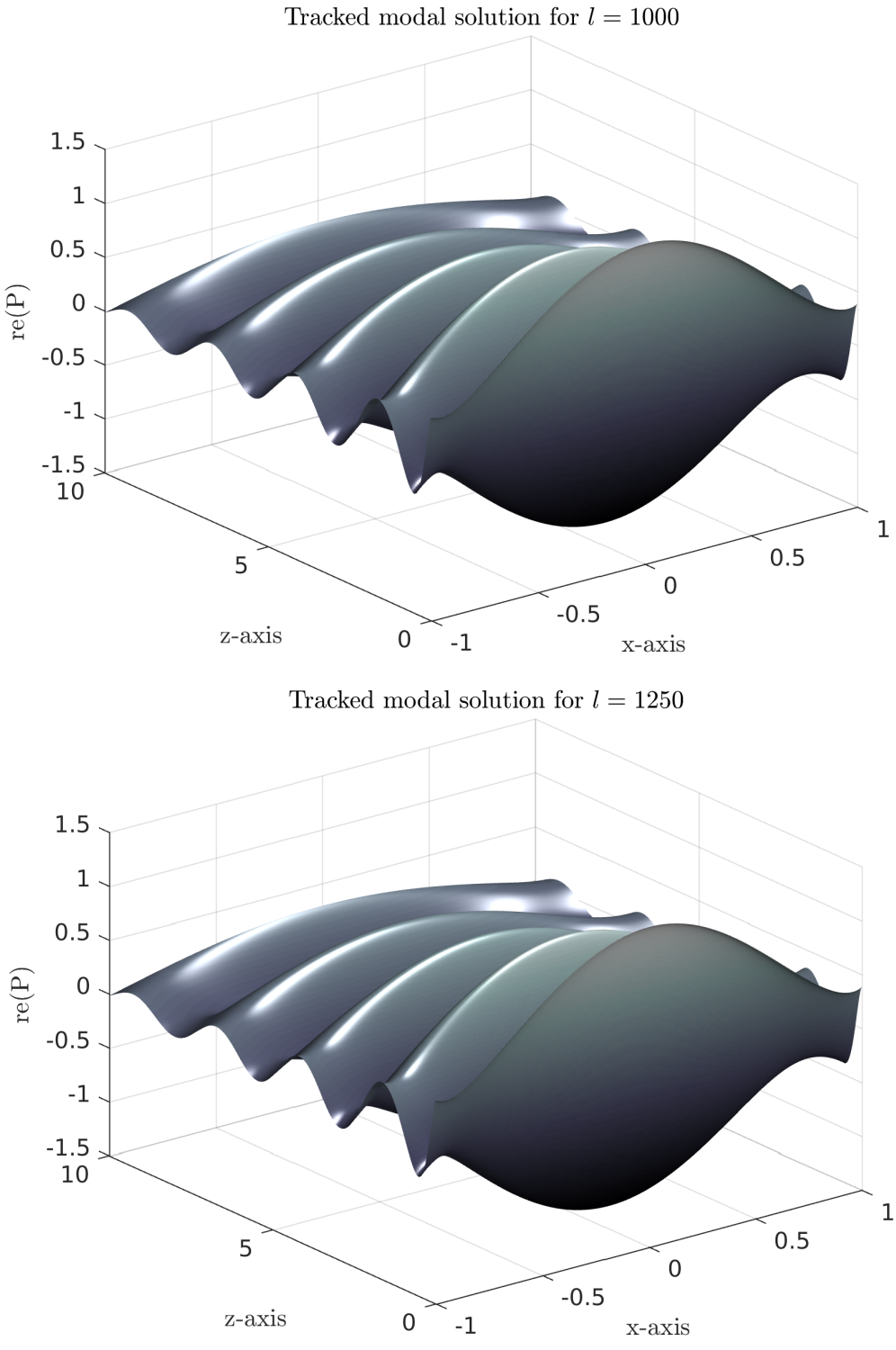


Figure 5.14: Modal solutions comparison for tracked roots within limits $[0, l], [-i, i]$

The two surface plots are graphically indistinguishable. The following figure is a surface plot of the difference between truncated modal solutions for $l = 1000$ and $l = 1250$.

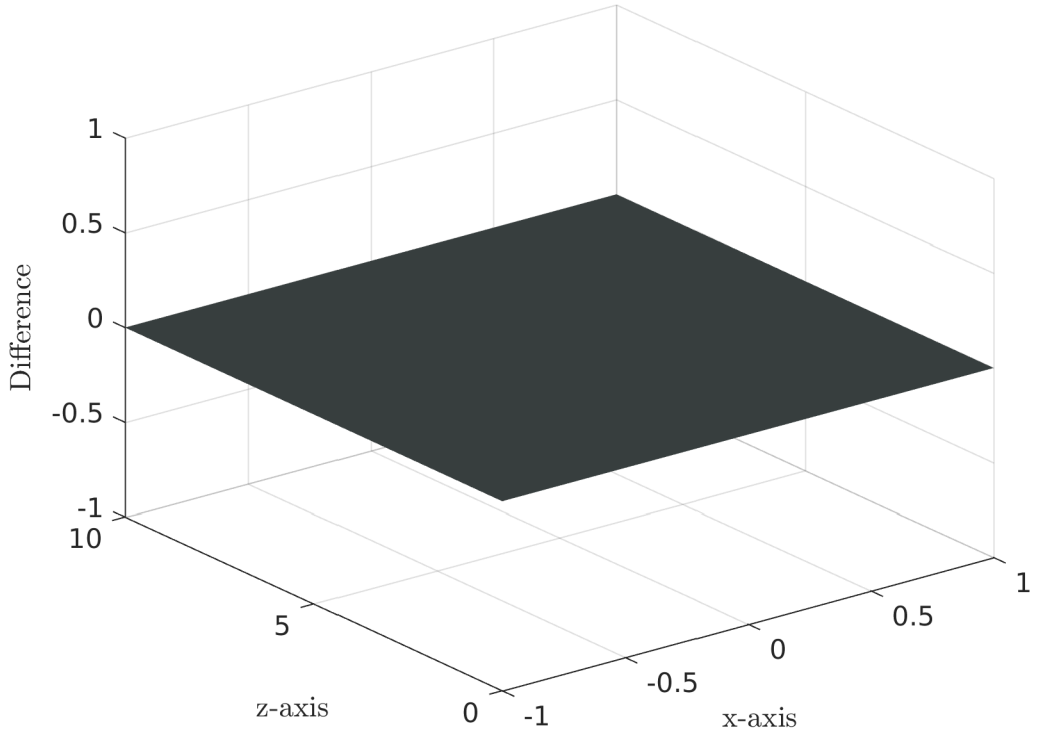


Figure 5.15: Difference of tracked modal solutions for $l = 1000$ and $l = 1250$

Again, Figure 5.15 shows a flat plane indicating no difference between modal solutions for $l = 1000$ and $l = 1250$. In support of this, the maximum difference between the two solutions is of order 10^{-7}. Therefore the truncated modal solution has sufficiently converged for $l = 1250$. The following figure provides the roots found for $l = 25$ and is produced by the `'boxes'` plotting capability of `roottracker`. We only display tracked roots up to $l = 25$ since roots after this value have unremarkable behaviour and quickly converge to the real line.

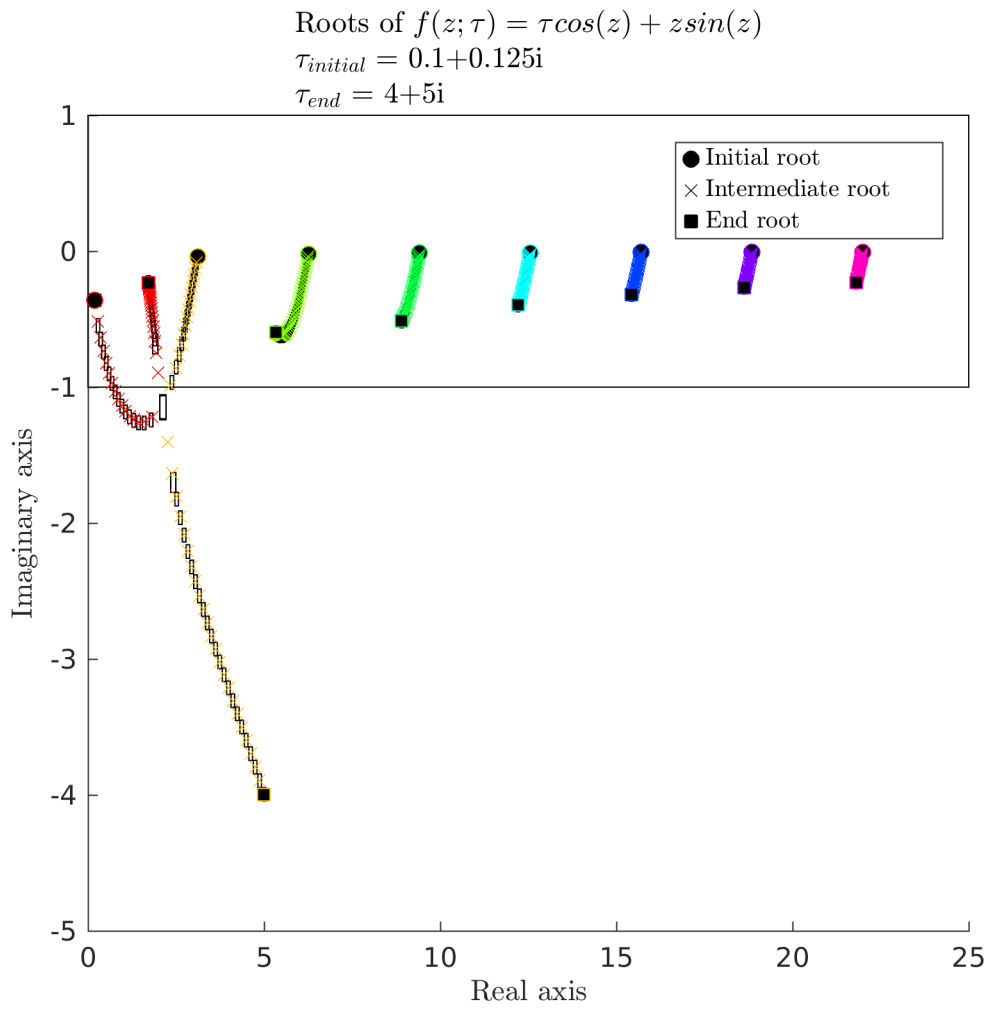


Figure 5.16: Roots found with tracking

Figure 5.16 shows a root leaves the initial search region, indicating the mode corresponding to this root is missing from the truncated modal solution without root tracking in Section 5.2.1. Let us again compare the exact boundary condition (2.10c) with the approximate modal solution obtained with root tracking; the comparison will be done with the same axis scaling as used in the untracked case (Figure 5.13).

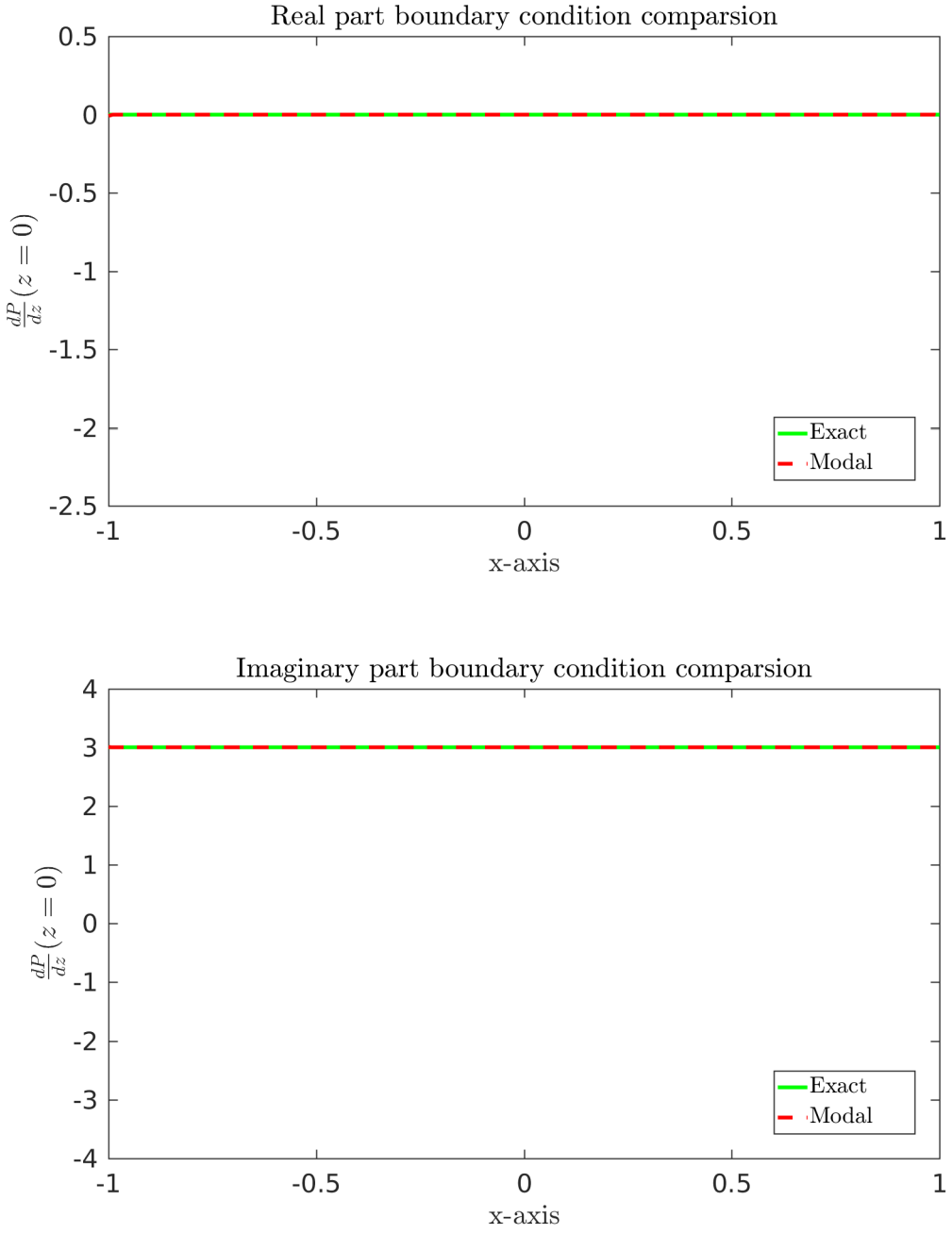


Figure 5.17: Boundary condition comparison for modal solution with tracking

As expected, the approximation satisfies the boundary condition with near perfect

accuracy.

The following figure is a surface plot of the difference between approximate modal solutions obtained with and without root tracking. Similarly to the example in Section 5.1, the z axis has been restricted to the range $[0, 2]$ to emphasis difference in solutions.

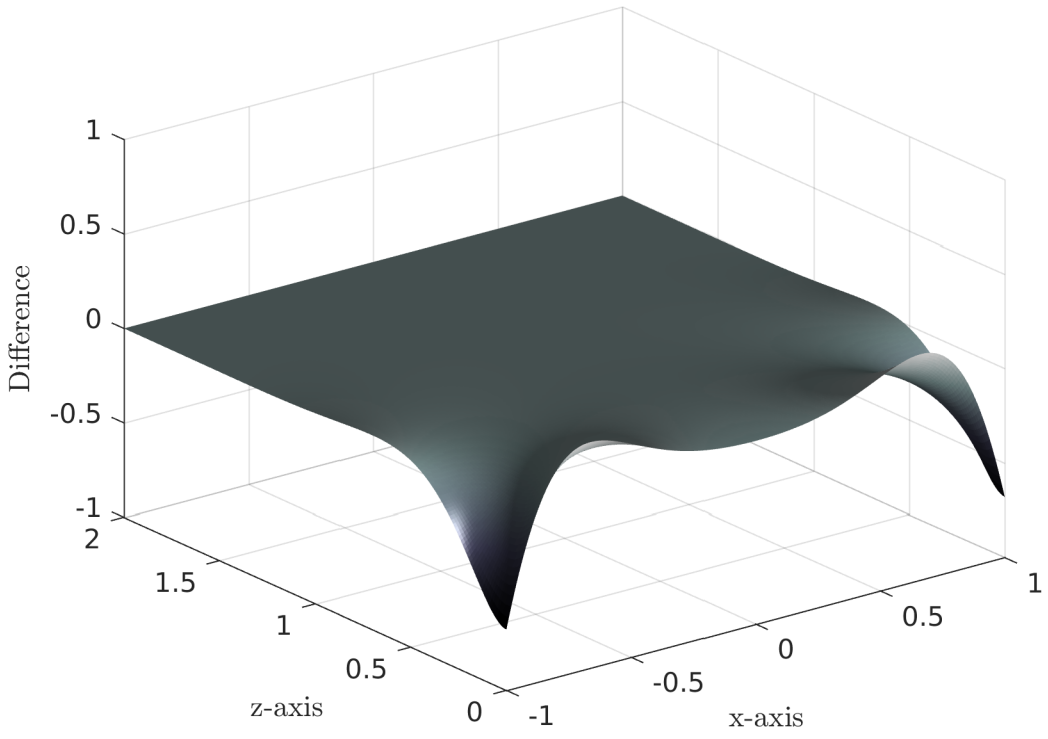


Figure 5.18: Difference of truncated modal solutions with and without root tracking

Figure 5.18 shows disparity between truncated modal solutions with and without root tracking. For such a case the importance of root tracking is conspicuous.

6 Summary and Concluding Remarks

Thales required further development to Franklin's root finder. Section 3.3.1 lists the updates made. Most notably:

- the functionality of multiple scripts (`WindNum`, `ArgWN`, `ArgWNSide`, `WNContourPA` and `WNChordContour`) has been condensed to a single script (`methodchoice`) which requires less memory and performs fewer operations;
- improved error checking;
- implementation of a plotting capability and
- improvements focused on human readability and usability of the code made, including reduced inputs to the root finder and renaming of variables and functions to more identifiable names.

The primary goal to extend the root finder to allow root tracking in response to changes in an implicit parameter has been achieved with great robustness and usability. Section 4.1 describes the developmental process and theory required to achieve this goal and Section 4.2 describes the theory's implementation in MATLAB, including the procedures set in place to ensure roots are effectively tracked.

Finally Section 5 provides two test cases for the representative non self-adjoint boundary value problem described in Section 2. In both cases, a truncated modal solution is determined with and without root tracking. By comparison with the boundary condition at the base of the duct, root tracking is shown to be sufficient in obtaining accurate truncated modal solutions, unlike those obtained without root tracking. Thus concluding the completion of the dissertation's goals.

Bibliography

[1] P. Cotterill and N. Willoughby, *Dispersion Equation for Semi-infinite Fluid-filled Rectangular Duct* (2016), acquired from Thales UK, March 2016.

[2] M. Crocker, *Handbook of Acoustics* (1998), Wiley-Interscience, p 84.

[3] A. Fokas, *Complex Variables: Introduction and Applications* (2013), Cambridge University Press, pp 260–261.

[4] D. Franklin, *An Investigation Into Methods of Determining the Roots of Dispersion Equations* (2014), master's dissertation submitted to the School of Mathematics, University of Manchester.

[5] V. Gibiat and F. Laloë, *Acoustical Impedance Measurements Using The Two Microphone Three Calibration Method* (1990), Journal of the Acoustical Society of America, Acoustical Society of America, Vol. 88, p 2533.

[6] H. Kuttruff, *Room Acoustics* (2014), 5th Edition, Taylor & Francis Group, p 37.

[7] A. Lars, *Complex Analysis: An Introduction to the Theory of Analytic Functions of One Complex Variable* (1979), McGraw-Hill Education, p 151.

[8] J. Lawrie, *Orthogonality Relations for Fluid-structural Waves in a Three-dimensional, Rectangular Duct with Flexible Walls* (2009), Proceedings of the Royal Society A, Vol. 465, pp 2347 – 2367.

[9] B. Otto, *Linear Algebra with Applications* (1995), Pearson Prentice Hall, Fourth Edition, pp 187 – 230.

[10] Lord J. Rayleigh, *Theory of Sound* (1877), London, Macmillan and co., Vol. 1.

[11] T. Schultz, L. Cattafesta III and M. Sheplak, *Modal Decomposition Method for Acoustic Impedance Testing in Rectangular Ducts* (2006), Journal of the Acoustical Society of America, Acoustical Society of America, Vol. 120, pp 3750 – 3758.

[12] H. Wilf, *A Global Bisection Algorithm for Computing the Zeros of Polynomials in the Complex Plane* (1978), Journal of the ACM Vol. 25, pp 415 – 420.

[13] N. Willoughby, *Investigations into a Complex Root-finder* (2016), acquired from Thales UK, February 2016.

Code Listings

ROOTTRACKER

```
1 %ROOTTRACKER
2 %   For a function "f" of a single explicit complex parameter "z" and
3 %   implicit parameter "tau", ROOTTRACKER finds initial roots of "f" for
4 %   "tau" = "tau_initial" in the complex region specified by "xsides" and
5 %   "ysides" by use of ROOTFINDER. ROOTTRACKER then increments "tau" by
6 %   "dtau" and finds the new roots. This is done by linearly approximating
7 %   how much each initial root will be incremented by and creating a small
8 %   search region around each of these approximations; ROOTFINDER then
9 %   finds the incremented roots in each search region. For sufficiently
```

```
10 %   small search regions and value of increments "dtau", only
   a single
11 %   incremented root should be found for each search region.
   If not, the
12 %   user is warned. Please note that "f" must be specified by
   SYMFUN and
13 %   must have its symbolic parameters specified by SYMS
   exactly as "z" and
14 %   "tau".
15 %
16 %
17 %   Minimial inputs:
18 %   r = roottracker(f, xsides, ysides, tau_initial, dtau,
   n_incr)
19 %
20 %   Maxmial inputs:
21 %
22 %   r = roottracker(f, xsides, ysides, tau_initial, dtau,
   n_incr,...
23 %                   root_box_size, 'plot')
24 %
25 %   INPUTS
26 %
27 %   "f":              A symbolic function of a single explicit
   parameter "z"
28 %                     and single implicit parameter "tau".
   These take
29 %                     complex values and are symbolic.
30 %
```

```
31 %   "xsides":        A length 2 vector giving the initial
   search region's
32 %                    real limits.
33 %
34 %   "ysides":        A length 2 vector giving the initial
   search region's
35 %                    imaginary limits.
36 %
37 %   "tau_initial":   The initial value of the implicit
   parameter "tau".
38 %
39 %   "dtau":          The amount "tau" is to be incremented by
   after each
40 %                    iteration.
41 %
42 %   "n_incr":        The number of incrementations to "tau".
43 %
44 %   OPTIONAL INPUTS
45 %
46 %   "root_box_size":       Optional parameter specifying the
   size of the
47 %                          incremented search regions. Default
    set to 0.1.
48 %
49 %   "'plot'" or "'boxes'": An optional input which must be
   given as the
50 %                          final input to ROOTTRACKER. This
   produces a
51 %                          scatter plot of roots found in the
   initial
```

```
52 %                          search region as well as all
   incremented roots.
53 %                          If "'boxes'" chosen then the
   initial search
54 %                          region and all incremented roots'
   search regions
55 %                          are plotted over the roots.
56 %
57 %   OUTPUTS
58 %
59 %   "r":              A vector containing the initial roots (1
   st row) and
60 %                     all incremented roots. The (j+1)th row
   contains the
61 %                     roots after j increments have been added
   to "tau". The
62 %                     ith column contains root i and all its
   increments.
63 %
64 %
65 %   Example:
66 %       syms z tau;
67 %       f = symfun(z + tau * cos(z * tau), [z tau]);
68 %       xsides = [-1,1];
69 %       ysides = [-1i,1i];
70 %
71 %       tau_initial = 1;
72 %       dtau = 0.1 + 0.1i;
73 %       n_incr = 8;
74 %
```

```
75 %       r = roottracker(f,xsides,ysides,tau_initial,dtau,
       n_incr,'plot');
76 %
77 %
78 %   Note that ROOTTRACKER requires the Symbolic Math Toolbox.
79 %
80 %   Developed by Louis Runcieman (louisruncieman@gmail.com).
81 %
82 %   See also ROOTFINDER, SYMS, SYMFUN.
83
84 function r = roottracker(f, xsides, ysides, tau_initial, dtau,
       n_incr, varargin)
85
86 global quiet;
87 global sub_n_incr; %If used, the number of sub-increments
       between increments. This must be
88                    %global so when "roottracker" recalls
                          itself the count is retained.
89                    %Automatically initialised as [].
90
91 %%%%% CHECKING ALL INPUTS CORRECTLY SPECIFIED.
92
93 if nargin > 9 %Checks correct number of inputs.
94     %Display messaged to inform user of error and to break out
           of program.
95     error('MATLAB:Invalid_inputs', 'Too many inputs to
          roottracker.')
96 end
97
```

```
98  if ~isa(f,'symfun') %Checks if f is specified as a symbolic
        function.
99      %Display messaged to inform user of error and to break out
            of program.
100     error('MATLAB:Invalid_inputs', ['Please specify "f" by
            SYMFUN, with SYMS "z" and "tau".\n',...
101             'Eg:\n\tsyms z tau \n\t', 'f = symfun(z^2 + tau * z
                + 1, [z tau]);\n'])
102 end
103
104 if ~isreal(xsides) || ~isnumeric(xsides) || length(xsides) ~=
        2 %Check xsides is numeric, real and length 2.
105     %Display messaged to inform user of error and to break out
            of program.
106     error('MATLAB:Invalid_inputs', 'Input "xsides" must be a
            length 2 vector with real entries.\n')
107 end
108
109 if real(ysides(1)) ~= 0 || real(ysides(2)) ~= 0 || ~isnumeric(
        ysides) || length(ysides) ~= 2 %Check ysides is numeric,
        purely imaginary and length 2.
110     %Display messaged to inform user of error and to break out
            of program.
111     error('MATLAB:Invalid_inputs', 'Input "ysides" must be a
            length 2 vector with purely imaginary entries.\n')
112 end
113
114 if xsides(1) >= xsides(2) || imag(ysides(1)) >= imag(ysides(2)
        ) %Checks if region nonempty.
```

```
115     %Display messaged to inform user of error and to break out
            of program.
116     error('MATLAB:Invalid_inputs', 'Empty initial search
            region specified.\n')
117 end
118
119 if ~isnumeric(tau_initial)
120     %Display messaged to inform user of error and to break out
            of program.
121     error('MATLAB:Invalid_inputs', 'Input "tau_initial" must
            be a complex number.\n')
122 end
123
124 if n_incr < 0 || rem(n_incr,1) ~= 0
125     %Display messaged to inform user of error and to break out
            of program.
126     error('MATLAB:Invalid_inputs', 'Input "n_incr" must be a
            nonnegative integer.\n')
127 end
128
129 if ~isnumeric(dtau)
130     %Display messaged to inform user of error and to break out
            of program.
131     error('MATLAB:Invalid_inputs', 'Input "dtau" must be
            numerical.\n')
132 end
133
134 if ~isempty(varargin) %Optional parameter checks and
    assignments.
135     if nargin == 7 %If one optional input specified.
```

```
136         if isnumeric(varargin{1}) %Check if optional input is
               numerical.
137             if varargin{1} > 0 && isreal(varargin{1}) %Check
                   whether "root_box_size" is positive real.
138                 root_box_size = varargin{1}; %User-specified
                       region size.
139                 plotting = 0; %No plot to be made.
140             else
141                 error('MATLAB:Invalid_inputs', 'User specified
                        "root_box_size" must be a positive real
                       number.')
142             end
143
144         elseif strcmpi(varargin{1}, 'plot') %If not numerical,
               check if it is defined as 'plot'.
145             root_box_size = 0.1; %Default region size.
146             plotting = 1; %Plot of roots without boxes.
147             figure
148             hold on
149
150         elseif strcmpi(varargin{1}, 'boxes')
151             root_box_size = 0.1; %Default region size.
152             plotting = 2; %Plot of roots with boxes.
153             figure
154             hold on
155
156         elseif strcmpi(varargin{1}, 'quiet')
157             root_box_size = 0.1;
158             plotting = 0;
159             quiet = 1;
```

```
160
161         else error('MATLAB:Invalid_inputs', ['Invalid optional
                inputs to ROOTTRACKER.\n',...
162                     'To plot tracked roots, specify final input
                        exactly as ''plot'' or ''boxes''.'])
163         end
164
165     elseif nargin == 8 %If two optional inputs specified.
166
167         if isnumeric(varargin{1}) && varargin{1} > 0 && isreal
                (varargin{1}) %Check whether "root_box_size" is
                numerical positive real.
168             root_box_size = varargin{1}; %User-specified
                    region size.
169             if strcmpi(varargin{2}, 'plot')
170                 plotting = 1;
171                 figure
172                 hold on
173
174             elseif strcmpi(varargin{2}, 'boxes')
175                 plotting = 2;
176                 figure
177                 hold on
178
179             elseif strcmpi(varargin{2}, 'quiet')
180                 plotting = 0;
181                 quiet = 1;
182             else error('MATLAB:Invalid_inputs', ['Invalid
                    optional inputs to ROOTTRACKER.\n',...
```

```
183                         'HHH or QUIET To plot tracked roots,
                               specify final input exactly as ''
                               plot'' or ''boxes''.'])
184             end
185         elseif strcmpi(varargin{1}, 'plot') && strcmpi(
               varargin{2}, 'quiet')
186             root_box_size = 0.1;
187             plotting = 1;
188             figure
189             hold on
190             quiet = 1;
191         elseif strcmpi(varargin{1}, 'boxes') && strcmpi(
               varargin{2}, 'quiet')
192             root_box_size = 0.1;
193             plotting = 2;
194             figure
195             hold on
196             quiet = 1;
197         else error('MATLAB:Invalid_inputs', ['Invalid optional
               inputs to ROOTTRACKER.\n',...
198                         'HHH or QUIET To plot tracked roots,
                               specify final input exactly as ''
                               plot'' or ''boxes''.'])
199         end
200
201     elseif nargin == 9 %All three optional inputs specified.
202         if isnumeric(varargin{1}) && varargin{1} > 0 && isreal
               (varargin{1}) %Check whether "root_box_size" is
               numerical positive real.
```

```
203             root_box_size = varargin{1}; %User-specified
                    region size.
204         else %First optional input must be "root_box_size".
205             error('MATLAB:Invalid_inputs', 'User specified "
                    root_box_size" must be a positive real number.'
                    )
206         end
207         if strcmpi(varargin{2}, 'plot')
208             plotting = 1; %Plot of roots without boxes.
209             figure
210             hold on
211         elseif strcmpi(varargin{2}, 'boxes')
212             plotting = 2; %Plot of roots without boxes.
213             figure
214             hold on
215         else error('MATLAB:Invalid_inputs', ['Invalid optional
                inputs to ROOTTRACKER.\n',...
216                 'HHH To plot tracked roots, specify final
                        input exactly as ''plot'' or ''boxes''.'
                        ])
217         end
218         if strcmpi(varargin{3}, 'quiet')
219             quiet = 1;
220         else error('MATLAB:Invalid_inputs', ['Invalid optional
                inputs to ROOTTRACKER.\n',...
221                 'HHH To quiet ''quiet''.'])
222         end
223     end
224 else
225     root_box_size = 0.1; %Default region size.
```

```
226     plotting = 0;
227 end %End checking isempty(varargin)
228 %%%%%End input checking.
229
230
231 syms z tau %Initialise the symbolic parameters of f. Note:
       these must be inputted exactly as z and tau.
232 dz(z, tau) = - diff(f, tau) * dtau / diff(f, z); %Linear
       approximation of increment in solution z as a function of z
        and tau.
233
234 [~, dz_denominator(z, tau)] = numden(dz); %The denominator of
       dz. Used to error check dividing by zero when approximating
        incremented root.
235 tau = tau_initial; %Set variable tau equal to initial value of
        tau.
236
237 %%%%% Initial roots before incrementation.
238 f_tau(z) = f(z,tau); %Remove depdency of tau from f and set
       tau = tau_initial (no longer variable).
239 r_temp = rootfinder(f_tau,xsides,ysides); %Find all the
       initial roots; r_temp(j) is the jth root.
240 if ~isempty(varargin) && strcmpi(varargin{end}, 'quiet')
241     %Reassign the "'quiet'" option if specified (since at the
           end of every call to rootfinder it is reset to []).
242     quiet = 1;
243 end
244 %r_temp is a vector containing all roots at a given increment
       and is overwritten after each incrementation of tau.
245
```

```
246 nroots = length(r_temp); %Number of roots to be tracked.
247 if isempty(quiet)
248     disp([sprintf('Number of roots found:\t'), num2str(nroots)
        ]) %Inform user how many roots are found.
249     disp('Tracking roots...') %Inform user roots will now be
        tracked.
250 end
251 r = NaN(n_incr + 1, nroots); %Final output of roottracker.
    Contains all roots for all increments.
252                              %The jth row corresponds to the
                                    roots at the (j-1)th increment
                                     of tau.
253 r(1,:) = r_temp; %First row contains the roots at tau =
    tau_initial.
254 %%%%% End initial roots.
255
256 %Temporary vectors containing information about the search
    region for each incremented root.
257 %These are overwritten after each incrementation of tau.
258 centre_temp = NaN(nroots,1); %Vector containing the centre
    point of each incremented root's search region.
259 xsides_temp = zeros(nroots, 2); %jth row is jth root's search
    region's real boundaries.
260 ysides_temp = zeros(nroots, 2); %jth row is jth root's search
    region's imaginary boundaries.
261
262 for j = 1:n_incr %Loop over all increments.
263     if isempty(quiet)
264         if isempty(sub_n_incr)
```

```
265             disp([sprintf('Number of increments added:\t'),
                    num2str(j)]) %Inform user how many increments
                    added.
266         else
267             disp([sprintf('Number of sub-increments added:\t')
                    , num2str(j)]) %Inform user how many sub-
                    increments added.
268         end
269     end
270
271     for k = 1:nroots %For the given increment, determine the
            incremented root's search region.
272         if ~isinf(r(j, k)) %If a root is inf then it is no
                longer being tracked.
273             %Linear approximation of new root value after
                    incrementing tau,
274             %serving as the centre of the new search region.
275
276             %Check if denominator of dz is (close to) zero at
                    current values of root and tau.
277             if abs(double(dz_denominator(r_temp(k), tau))) <
                    0.00001
278                 centre_temp(k) = double(r_temp(k)); %Centre of
                        increment root's search region is given
                        the value of the previous root.
279             else %Otherwise, add a linear approximation dz to
                    previous root.
280                 centre_temp(k) = double(r_temp(k) + dz(r_temp(
                        k), tau));
281             end
```

```
282             %Search region real and imaginary limits
                   respectively. Region is a square of size
                   root_box_size.
283             xsides_temp(k,:) = [real(centre_temp(k)) - 0.5 *
                   root_box_size, real(centre_temp(k)) + 0.5 *
                   root_box_size];
284             ysides_temp(k,:) = [imag(centre_temp(k)) - 0.5 *
                   root_box_size, imag(centre_temp(k)) + 0.5 *
                   root_box_size] * 1i;
285         end
286     end
287
288     tau = tau + dtau; %Update tau.
289     f_tau(z) = f(z, tau); %Update f for new value of tau and
           remove depedency on tau by assigning f_tau(z).
290
291     for k = 1:nroots %For the given increment, calculate each
           root's incremented value.
292         if isinf(r(j, k)) %If the root is inf, then it is no
               longer tracked.
293             r_temp(k) = inf; %Subsequent increments of an
                   untracked root are inf also.
294         else
295             r_temp2 = rootfinder(f_tau, xsides_temp(k,:),
                   ysides_temp(k,:)); %Find each roots value after
                    increment added.
296             if ~isempty(varargin) && strcmpi(varargin{end}, '
                   quiet')
297                 %Reassign the "'quiet'" option if specified (
                       since at the end of every call to
```

```
                rootfinder it is reset to []).
298             quiet = 1;
299         end
300         %r_temp2 contains the kth root after j
               incrementations of tau.
301         %It is overwritten for each root at the jth
               incrementation.
302
303         increases = 0; %Initialise number of times the
               incremented root's search
304                        %region has been increased in size
                              for the case of a missing root.
305
306         while isempty(r_temp2) %Case of incremented root
               missing from it's search region.
307
308             increases = increases + 1; %Number of times
                   increment search region increased.
309             if isempty(quiet)
310                 if increases == 1 %Inform user the first
                       time the search region will be
                       increased.
311                     disp([sprintf('\nIncreasing increment
                            region size for root '), num2str(
                           k), sprintf('...\n')])
312                 end
313             end
314             %Increase the size of the region by 5%.
315             xsides_temp(k,:) = [xsides_temp(k,1) - 5  /
                   100 * abs(xsides_temp(k,1) - xsides_temp(k
```

```
                    ,2)), xsides_temp(k,2) + 5  / 100 * abs(
                    xsides_temp(k,1) - xsides_temp(k,2))];
316                 ysides_temp(k,:) = [ysides_temp(k,1) - 5i /
                    100 * abs(ysides_temp(k,1) - ysides_temp(k
                    ,2)), ysides_temp(k,2) + 5i / 100 * abs(
                    ysides_temp(k,1) - ysides_temp(k,2))];
317
318                 r_temp2 = rootfinder(f_tau,xsides_temp(k,:),
                    ysides_temp(k,:)); %Recalculate root.
319                 if ~isempty(varargin) && strcmpi(varargin{end
                    }, 'quiet')
320                     %Reassign the "'quiet'" option if
                        specified (since at the end of every
                        call to rootfinder it is reset to []).
321                     quiet = 1;
322                 end
323
324                 if increases == 6 %~34% increase to increment
                    search region (compound increase).
325
326                     %Using sub-increments to determine missing
                         root.
327                     if isempty(sub_n_incr) %Before any sub-
                        increments have been used.
328
329                         if isempty(quiet)
330                             %Inform user and ask if sub-
                                increments should be used to
                                find the root.
```

```
331                     disp(sprintf(['\nIncrement region
                           size increased by maximum
                           amount.\n',...
332                                 '(Note: consider
                                    decreasing "dtau"
                                     or increasing "
                                    root_box_size").\
                                    n']))
333                 end
334                 %Include sub-incrememements to determine
                        root.
335                 %Specify a small search region around
                       the kth incremented (ie. previous)
                       root.
336                 %To find the (k+1)th root, we divide
                       dtau by sub_n_incr and add
                       sub_n_incr increments of
337                 %this to the value of tau at the
                       previous root, thus finding the
                       root at tau + dtau.
338                 %This is done for sub_n_incr = 10, 20,
                        30 and after this the root will no
                        longer be tracked.
339
340                 sub_n_incr = 10; %Add 10 sub-
                       increments between tau and tau +
                       dtau.
341
342                 %Small region around previous root.
```

```
343                         centre_temp2 = double(r_temp(k)); %
                               Centre of increment root's search
                               region is given the value of the
                               previous root.
344                         xsides_temp2 = [real(centre_temp2) -
                               0.5 * root_box_size, real(
                               centre_temp2) + 0.5 * root_box_size
                               ];
345                         ysides_temp2 = [imag(centre_temp2) -
                               0.5 * root_box_size, imag(
                               centre_temp2) + 0.5 * root_box_size
                               ] * 1i;
346
347                         if isempty(quiet)
348                             disp(sprintf('Including sub-
                                   increments...\n\n***\t10\tsub-
                                   increments'))
349                         end
350                         %Track from previous root.
351                         r_temp3 = roottracker(f, xsides_temp2,
                                ysides_temp2, tau - dtau, dtau /
                               sub_n_incr, sub_n_incr,
                               root_box_size);
352                         if ~isempty(varargin) && strcmpi(
                               varargin{end}, 'quiet')
353                             %Reassign the "'quiet'" option if
                                   specified (since at the end of
                                   every call to roottracker it is
                                    reset to []).
354                             quiet = 1;
```

```
355                         end
356                         if isempty(quiet)
357                             disp(sprintf('***\n'))
358                         end
359                             while isnan(r_temp3)
360                             if sub_n_incr == 10 || sub_n_incr
                                   == 20
361                                  sub_n_incr = sub_n_incr + 10;
362                                  if isempty(quiet)
363                                      disp(sprintf(['***\t',
                                            num2str(sub_n_incr), '\
                                            tsub-increments']))
364                                  end
365                                  r_temp3 = roottracker(f,
                                     xsides_temp2, ysides_temp2,
                                      tau - dtau, dtau /
                                     sub_n_incr, sub_n_incr,
                                     root_box_size);
366                                  if ~isempty(varargin) &&
                                     strcmpi(varargin{end}, '
                                     quiet')
367                                      %Reassign the "'quiet'"
                                         option if specified (
                                         since at the end of
                                         every call to
                                         roottracker it is reset
                                          to []).
368                                      quiet = 1;
369                                  end
370                                  if isempty(quiet)
```

```
371                                 disp(sprintf('***\n'))
372                             end
373                         elseif sub_n_incr == 30
374                             if isempty(quiet)
375                                 disp(sprintf('Maximum
                                        number of subincrements
                                         added for this root.\
                                        nNo longer tracking
                                        root.\n'))
376                             end
377                             r_temp3 = inf;
378                             sub_n_incr = [];
379                         end
380                     end
381                     if size(r_temp3, 2) ~= 1 %Output error
                         if more than one root found in sub
                        -increment root's search region.
382                         if isempty(quiet)
383                             disp(sprintf('Multiple roots
                                found sub-increment region
                                .\n'))
384                         end
385                         r_temp3 = inf; %No longer track
                            root. %Could have this as NaN
                            and use more increments...
386                     end
387                     r_temp2 = r_temp3(end);
388
389                 else %This else is only met if sub-
                    increments have already been used
```

```
390                         %(after roottracker has been
                               internally called).
391                         r = NaN; %This is only given to the
                               value r_temp3, meaning more sub-
                               increments are required.
392                     return %Return to roottracker after
                           roottracker was interally called.
393                 end %end usage of sub-increments.
394             end %End if increases == 6.
395         end %End case of missing root.
396
397
398         if length(r_temp2) > 1 %Case of multiple roots in
               incremented root's search region.
399
400             %Inform user. Gives parameters so that user
                   can rerun roottracker to track these
                   multiple roots.
401             if isempty(quiet)
402                 disp(sprintf(['\nWarning: either a double
                       root has split into simple roots or an\
                       n'...
403                             'incremented root has passed
                                 near another root.
                               Consider rerunning the\n'
                               , ...
404                             'program with "tau_initial"
                               > ', num2str(tau), ' (the
                                current value of "tau")\
                               n',...
```

```
405                                     '"xsides" = [', num2str(
                                           xsides_temp(k,1)), ', ',
                                           num2str(xsides_temp(k,2))
                                           ,...
406                                     '] and ysides = [', num2str(
                                           imag(ysides_temp(k,1))),
                                           'i, ', num2str(imag(
                                           ysides_temp(k,2))), 'i].\
                                           n\n',...
407                                     'This root will no longer be
                                            tracked.']))
408                 end
409                 r_temp2 = inf; %Set root value to inf and no
                       longer track root.
410             end %End case of multiple roots in incremented
                   root's search region.
411
412             r_temp(k) = r_temp2; %Store all incremented root's
                    values in temporary vector.
413         end %End check if previous root was inf.
414     end %End cycling k for roots at jth incrementation of tau.
415
416     %Permanently store all roots at the jth incrementation.
417     r(j+1,:) = r_temp; %(j+1)th contains roots after j
           increments added.
418
419     %If 'boxes' specified, plot incremented roots' boxes (
           search regions).
420     if plotting == 2
421         for k = 1:nroots
```

```
422             rectangle('position', [xsides_temp(k,1), imag(
                    ysides_temp(k,1)), xsides_temp(k,2) -
                    xsides_temp(k,1), imag(ysides_temp(k,2) -
                    ysides_temp(k,1))]); %Specified region.
423         end
424         rectangle('position', [xsides(1), imag(ysides(1)),
                xsides(2) - xsides(1), imag(ysides(2) - ysides(1))
                ]); %Initial search region.
425     end
426 end
427
428 sub_n_incr = []; %Resets counter of number of sub-increments
    used.
429 %This resets the counter after each time sub-increments have
    been used for
430 %a single call of roottracker at the command line.
431 %This also resets the counter when the initial call of
    roottracker
432 %terminates, otherwise the value of "sub_n_incr" would be
    retained if
433 %calling roottracker twice at the command line without clear
    the global
434 %workspace.
435 quiet = [];
436
437 %If "'plot'" or "'boxes'" specified, plot all incrementations
    of roots.
438 if plotting == 1 || plotting == 2
439
```

```
440     if n_incr > 1 %Fudge used to plot legend. Needs to be
            before main plots to keep colours.
441         h = zeros(3,1); %%%LEGEND HHH
442         h(1) = plot(real(r(1,1)), imag(r(1,1)),'ko', '
                MarkerFaceColor', 'k');
443         h(2) = plot(real(r(2,1)), imag(r(2,1)),'kx');
444         h(3) = plot(real(r(1,1)), imag(r(1,1)),'ks', '
                MarkerFaceColor', 'k');
445         legend(h,{'Initial root','Intermediate root', 'End
                root'}, 'Interpreter','LaTex');
446     else
447         h = zeros(2,1); %%%LEGEND HHH
448         h(1) = plot(real(r(1,1)), imag(r(1,1)),'ko', '
                MarkerFaceColor', 'k');
449         h(2) = plot(real(r(1,1)), imag(r(1,1)),'ks', '
                MarkerFaceColor', 'k');
450         legend(h,{'Initial root','End root'}, 'Interpreter','
                LaTex');
451     end
452
453     rootcolour = hsv(nroots); %Produce unique colour map for
            each root and its incrementations.
454     for j = 1: n_incr + 1 %Plus 1 because we have the initial
            roots also.
455         for k = 1:nroots
456             if j == 1 %Initial root marked by circle.
457                 plot(real(r(1,k)), imag(r(1,k)), 'Color',
                        rootcolour(k,:), 'Marker', 'o', '
                        MarkerFaceColor', 'k')%rootcolour(k,:))
```

```
458             elseif j == n_incr + 1 %Final root marked by
                    square.
459                 plot(real(r(n_incr + 1,k)), imag(r(n_incr + 1,
                        k)), 'Color', rootcolour(k,:), 'Marker', 's
                        ', 'MarkerFaceColor', 'k')%rootcolour(k,:))
460             else %Middle roots marked by cross.
461                 plot(real(r(j,k)), imag(r(j,k)), 'Color',
                        rootcolour(k,:), 'Marker', 'x')
462             end
463         end
464     end
465
466
467     xlabel('Real axis', 'Interpreter', 'LaTex'); %Label for x
            axis.
468     ylabel('Imaginary axis', 'Interpreter', 'LaTex'); %Label
            for y axis.
469     syms tau;
470     TeXstring = texlabel(f(z, tau));
471     titlestring = strcat('$', TeXstring, '$');
472     title({['Roots of $f(z; \tau) = $' , ' ', titlestring], ['
            $\tau_{initial} = $', ' ', num2str(tau_initial)], ['$\
            tau_{end}$ = ', ' ', num2str(tau_initial + n_incr *
            dtau)]}, 'Interpreter', 'LaTex'); %Title.
473     %title({[strcat('f(z; tau)=', char(f))], ['$\tau_{initial}
             = $', num2str(tau_initial)], ['$\tau_{end}$ = ',
            num2str(tau_initial + n_incr * dtau)]}, 'Interpreter','
            LaTex'); %Title.
474     hold off
475 end
```

ROOTFINDER

```
%ROOTFINDER
%   ROOTFINDER finds all roots (and multiplicities) of a function of
%   one complex parameter in a region of the complex plane. This is done by
%   calculating the winding number of the contour travelling around the
%   region's boundaries. If the winding number is greater than 4 the region
%   is quadsectioned into 4 subregions and this process is repeated until
%   every subregion has a winding number less than or equal to 4 (if the
%   winding number is greater than 4 after a given number of quadsections
%   it is assumed the subregion has a single root with multiplicity equal
%   to the winding number).
%   At this stage, a principle argument integral weighted with z^n is
%   calculated (where n is the winding number). This integral can be
%   written as a polynomial of order n, which is solved exactly using
%   quadratic, cubic and quartic formulae (hence the need for the winding
%   number to be less than or equal to 4). The solutions of these
```

```
16 %   polynomials are the roots of the function.
17 %   If a root is near a boundary the region is increased by 1%
       in every
18 %   direction. If quadsectioning is required and a root is
       near an inner
19 %   boundary the centre of the quadsection is relocated.
20 %   The region should not cross branch cuts or contain poles.
21 %
22 %   Minimal inputs and outputs:
23 %   r = ROOTFINDER(f, xsides, ysides)
24 %
25 %   Maximal inputs and outputs:
26 %   [r, WN] = ROOTFINDER(f, xsides, ysides, 'chords',
       numchords, 'plot')
27 %
28 %   INPUTS
29 %
30 %   "f":          A symbolic function of a single symbolic
       parameter "z".
31 %
32 %   "xsides":     A length 2 vector giving the search region's
       real limits.
33 %
34 %   "ysides":     A length 2 vector giving the search region's
       imaginary
35 %                 limits.
36 %
37 %   OPTIONAL INPUTS
38 %
```

```
39 %   'chords':    Please see METHODCHOICE. If not specified,
   method 0
40 %                selected. If specified, method 1 selected.
41 %                Method 0: Determine winding number by Cauchy'
   s argument
42 %                          principle. This requires the
   program to
43 %                          calculate the symbolic derivative
   of "f".
44 %                Method 1: Determine winding number by
   tracking change in
45 %                          argument of "f". This requires each
    edge of the
46 %                          region to be divided into chords,
   meaning
47 %                          "numchords" must be specified.
48 %
49 %   "numchords": Should 'chords' be specified, this positive
   integer
50 %                specifies the number of chords each edge of
   the region is
51 %                to be divided into. This input is optional
   and can be
52 %                omitted when using method 0 (although this is
    not
53 %                necessary). Please note, if value given is
   insufficiently
54 %                small it is possible not not all roots will
   be found.
55 %
```

```
56 %   "'plot'":    An optional input which must be given as the
   final input
57 %                to ROOTFINDER. This produces a scatter plot
   of all roots
58 %                in the search region.
59 %
60 %   "'quiet'":
61 %
62 %   OUTPUTS
63 %
64 %   "r":         Vector containing the approximate (complex)
   roots of "f".
65 %
66 %   "WN":        Optional output giving the winding number of
   the region.
67 %
68 %
69 %   Example:
70 %       syms z;
71 %       f = symfun(z^2 + 2, z);
72 %       xsides = [-1, 1];
73 %       ysides = [-2i, 2i];
74 %
75 %       r = rootfinder(f,xsides,ysides,'plot');
76 %
77 %
78 %   Note that ROOTFINDER requires the Symbolic Math Toolbox.
79 %
80 %   Developed by Louis Runcieman (louisruncieman@gmail.com)
   adapted from
```

```
81  %   code written by David Franklin (davechrisfranklin@hotmail.
       co.uk).
82  %
83  %   See also METHODCHOICE, QUADSEC, ARGCHANGE, POLYSOLVER,
       QUADGK, SOLVE,
84  %            SYMS, SYMFUN.
85
86  function [r, varargout] = rootfinder(f, xsides, ysides,
       varargin)
87
88  global quiet
89  method = 0;
90  numchords = []; %Default numchords as empty to satisfy
       variable inputs for subfunctions of rootfinder.
91  plotting = [];
92  %%%%% CHECKING ALL INPUTS CORRECTLY SPECIFIED.
93  if nargin > 7 %Checks correct number of inputs.
94      %Display messaged to inform user of error and to break out
           of program.
95      error('MATLAB:Invalid_inputs', 'Too many inputs to
          rootfinder.')
96  end
97
98  %if ~isa(f,'symfun') %Checks if f is specified as a symbolic
       function.
99  %     %Display messaged to inform user of error and to break
       out of program.
100 %     error('MATLAB:Invalid_inputs', ['Please specify "f" by
       symfun. Eg:\n',...
```

```
101 %           'syms z\n', 'f = symfun(z^2 + 1, z)\n', 'rootfinder
    (f,xsides,ysides,method)'])
102 %end
103
104 if ~isreal(xsides) || ~isnumeric(xsides) || length(xsides) ~=
    2 %Check xsides is numeric, real and length 2.
105     %Display messaged to inform user of error and to break out
            of program.
106     error('MATLAB:Invalid_inputs', ['Input "xsides" must be a
        length 2 vector with real entries:\n'...
107             'rootfinder(f,xsides,ysides,method)'])
108 end
109
110 if real(ysides(1)) ~= 0 || real(ysides(2)) ~= 0 || ~isnumeric(
    ysides) || length(ysides) ~= 2 %Check ysides is numeric,
    purely imaginary and length 2.
111     %Display messaged to inform user of error and to break out
            of program.
112     error('MATLAB:Invalid_inputs', ['Input "ysides" must be a
        length 2 vector with purely imaginary entries:\n'...
113             'rootfinder(f,xsides,ysides,method)'])
114 end
115
116 if xsides(1) >= xsides(2) || imag(ysides(1)) >= imag(ysides(2)
    ) %Checks if region nonempty.
117     %Display messaged to inform user of error and to break out
            of program.
118     error('MATLAB:Invalid_inputs', ['Empty search region:\n'
        ...
119             'rootfinder(f,xsides,ysides,method)'])
```

```
120 end
121
122 if ~isempty(varargin)
123     if nargin == 4 %One optional input.
124         if strcmpi(varargin{1}, 'plot')
125             plotting = 1;
126         elseif strcmpi(varargin{1}, 'quiet')
127             quiet = 1;
128         else
129             error('MATLAB:Invalid_inputs', ['Invalid optional
                    inputs to rootfinder:\n',...
130                 'rootfinder(f, xsides, ysides, ''chords'',
                        numchords, ''plot'')'])
131         end
132     elseif nargin == 5 %Two optional inputs specified.
133         if strcmpi(varargin{1}, 'chords')
134             if isnumeric(varargin{2}) && varargin{2} > 0 &&
                    rem(varargin{2},1) == 0 %Check numchords entry
                    is nonempty and is positive integer.
135                 disp('Applying ''chords'' method...')
136                 method = 1;
137                 numchords = varargin{2};
138                 err = 0;
139             else
140                 error('MATLAB:Invalid_inputs', ['When choosing
                        ''chords'' method, must have positive
                        integer "numchords" specified:\n',...
141                     'rootfinder(f,xsides,ysides,''chords'',
                            numchords).'])
142             end
```

```
143         elseif strcmpi(varargin{1}, 'plot') && strcmpi(
                varargin{2}, 'quiet')
144             plotting = 1;
145             quiet = 1;
146         else
147             error('MATLAB:Invalid_inputs', ['Invalid optional
                    inputs to rootfinder:\n',...
148                     'rootfinder(f, xsides, ysides, ''chords'',
                            numchords, ''plot'')'])
149         end
150     elseif nargin == 6; %3 optional inputs.
151         if strcmpi(varargin{1}, 'chords') && isnumeric(
                varargin{2}) && varargin{2} > 0 && rem(varargin
                {2},1) == 0 %Check numchords entry is nonempty and
                is positive integer.
152             method = 1;
153             numchords = varargin{2};
154             err = 0;
155             if strcmpi(varargin{3}, 'plot')
156                 disp('Applying ''chords'' method...')
157                 plotting = 1;
158             elseif strcmpi(varargin{3}, 'quiet')
159                 quiet = 1;
160             else
161                 error('MATLAB:Invalid_inputs', ['Invalid
                        optional inputs to rootfinder:\n',...
162                         'rootfinder(f, xsides, ysides, ''chords'
                                ', numchords, ''plot'')'])
163             end
164         else
```

```
165             error('MATLAB:Invalid_inputs', ['Invalid optional
                    inputs to rootfinder:\n',...
166                     'rootfinder(f, xsides, ysides, ''chords'',
                            numchords, ''plot'')']);
167         end
168     elseif nargin == 7; %4 optional inputs.
169         if strcmpi(varargin{1}, 'chords') && isnumeric(
                varargin{2}) && varargin{2} > 0 && rem(varargin
                {2},1) == 0 && strcmpi(varargin{3}, 'plot') &&
                strcmpi(varargin{4}, 'quiet')
170             method = 1;
171             numchords = varargin{2};
172             err = 0;
173             plotting = 1;
174             quiet = 1;
175         else
176             error('MATLAB:Invalid_inputs', ['Invalid optional
                    inputs to rootfinder:\n',...
177                     'rootfinder(f, xsides, ysides, ''chords'',
                            numchords, ''plot'')']);
178         end
179     else error('MATLAB:Invalid_inputs', ['Too many optional
            inputs to rootfinder:\n',...
180                 'rootfinder(f, xsides, ysides, ''chords'',
                        numchords, ''plot'')']);
181     end
182 end
183 %%%%% INPUT CHECK COMPLETE.
184
```

```
185 if double(f == 0) && xsides(1) < 0 && 0 < xsides(2) && imag(
    ysides(1)) < 0 && 0 < imag(ysides(2))
186     r = 0;
187     varargout{1} = 1;
188     return
189 end
190
191 [WN, err] = methodchoice(f, xsides, ysides, method, numchords)
    ; %Find winding number.
192
193 %Check to see if a root lies on the edge of the region. If so,
    increase
194 %region by 1% in every direction.
195 while abs(real(WN) - round(real(WN))) > 0.0001 || abs(imag(WN)
    ) > 0.0001 || isnan(imag(WN)) || err > 0
196
197     %Increase the size of the region by 1%.
198     xsides = [xsides(1) - 1   / 100 * abs(xsides(1) - xsides(2)
        ), xsides(2) + 1   / 100 * abs(xsides(1) - xsides(2))];
199     ysides = [ysides(1) - 1i / 100 * abs(ysides(1) - ysides(2)
        ), ysides(2) + 1i / 100 * abs(ysides(1) - ysides(2))];
200
201     [WN, err] = methodchoice(f, xsides, ysides, method,
        numchords); %Recalculate winding number.
202
203     %Display message informing user of boundary change.
204     if isempty(quiet)
205         disp('A root near boundary: region has been increased
            by 1%.')
206     end
```

```
207 end
208
209
210 if WN >= 0.9999 && WN <= 4.00001 %Check if winding number is
    between 1 and 4.
211
212     WI = zeros(1,round(WN)); %Initialise winding integral
        values.
213
214     for n = 1:round(WN) %Calculate winding integrals.
215         WI(n) = methodchoice(f, xsides, ysides, 2, n); %
            Applies quadgk algorithm to find winding integral.
216     end
217
218     r = double(polysolver(WI, WN)); %Solve resulting
        polynomials.
219
220 elseif WN > 4.00001 %If winding number greater than 4, apply
    quadsectioning.
221     if isempty(quiet)
222         disp('Quadsectioning region...')
223     end
224     r = zeros(1,round(WN)); %Initialise vector containing
        roots.
225     rn = 1; %Define the initial root number.
226     quadnum = 1; %Initialise quadnum, counts number of
        quadsections.
227     r = quadsec(f, xsides, ysides, r, rn, quadnum, method,
        numchords); %Apply quadsection.
228
```

```
229 elseif abs(WN) < 0.00001 %Zero winding number, therefore no
    roots in region.
230     if isempty(quiet)
231         disp('No roots found in the specified region.')
232     end
233     r = [];
234     varargout{1} = 0;
235     return
236 else
237     error('Negative winding number.')
238 end
239
240 %HHH need to comment these two out when using TestFind.
241 [~, idx] = sort(real(r)); %Sort roots by real part increasing.
242 r = r(idx);
243
244 varargout{1} = round(WN); %Optionally outputs the winding
    number (which is rounded).
245
246
247
248 %Plotting capability.
249 if plotting == 1 %If specified give plot of roots in region.
250     if isempty(quiet)
251         disp('Plotting...')
252     end
253     nroots = size(r, 2);
254     rootcolour = hsv(nroots); %Produce colour map. Each root
        is given a unique colour.
255     figure
```

```
256     rectangle('position', [xsides(1), imag(ysides(1)), xsides
           (2) - xsides(1), imag(ysides(2) - ysides(1))]); %Search
            region (includes any size increases).
257     hold on
258     for i=1:nroots
259         plot(real(r(i)), imag(r(i)), 'Color', rootcolour(i,:),
               'Marker', 'x'); %Plot each root.
260     end
261     xlabel('Real axis','Interpreter','LaTex'); %Label for x
           axis.
262     ylabel('Imaginary axis','Interpreter','LaTex'); %Label for
            y axis.
263     TeXstring = texlabel(f); %Label in LaTeX format.
264     title(['Roots of $f(z) = $' , ' ', TeXstring],'Interpreter
           ','LaTex'); %Title.
265     hold off
266 end
267
268 quiet = [];
269 end %End rootfinder.
```

METHODCHOICE

```
1 %ROOTFINDER
2 %   ROOTFINDER finds all roots (and multiplicities) of a
     function of
3 %   one complex parameter in a region of the complex plane.
     This is done by
4 %   calculating the winding number of the contour travelling
     around the
```

```
5  %   region's boundaries. If the winding number is greater than
       4 the region
6  %   is quadsectioned into 4 subregions and this process is
   repeated until
7  %   every subregion has a winding number less than or equal to
       4 (if the
8  %   winding number is greater than 4 after a given number of
   quadsections
9  %   it is assumed the subregion has a single root with
   multiplicity equal
10 %   to the winding number).
11 %   At this stage, a principle argument integral weighted with
       z^n is
12 %   calculated (where n is the winding number). This integral
   can be
13 %   written as a polynomial of order n, which is solved
   exactly using
14 %   quadratic, cubic and quartic formulae (hence the need for
   the winding
15 %   number to be less than or equal to 4). The solutions of
   these
16 %   polynomials are the roots of the function.
17 %   If a root is near a boundary the region is increased by 1%
       in every
18 %   direction. If quadsectioning is required and a root is
   near an inner
19 %   boundary the centre of the quadsection is relocated.
20 %   The region should not cross branch cuts or contain poles.
21 %
22 %   Minimal inputs and outputs:
```

```
23 %   r = ROOTFINDER(f, xsides, ysides)
24 %
25 %   Maximal inputs and outputs:
26 %   [r, WN] = ROOTFINDER(f, xsides, ysides, 'chords',
   numchords, 'plot')
27 %
28 %   INPUTS
29 %
30 %   "f":          A symbolic function of a single symbolic
   parameter "z".
31 %
32 %   "xsides":     A length 2 vector giving the search region's
   real limits.
33 %
34 %   "ysides":     A length 2 vector giving the search region's
   imaginary
35 %                 limits.
36 %
37 %   OPTIONAL INPUTS
38 %
39 %   'chords':     Please see METHODCHOICE. If not specified,
   method 0
40 %                 selected. If specified, method 1 selected.
41 %                 Method 0: Determine winding number by Cauchy'
   s argument
42 %                           principle. This requires the
   program to
43 %                           calculate the symbolic derivative
   of "f".
```

```
44 %                Method 1: Determine winding number by tracking change in
45 %                          argument of "f". This requires each edge of the
46 %                          region to be divided into chords, meaning
47 %                          "numchords" must be specified.
48 %
49 %   "numchords": Should 'chords' be specified, this positive integer
50 %                specifies the number of chords each edge of the region is
51 %                to be divided into. This input is optional and can be
52 %                omitted when using method 0 (although this is not
53 %                necessary). Please note, if value given is insufficiently
54 %                small it is possible not not all roots will be found.
55 %
56 %   "'plot'":    An optional input which must be given as the final input
57 %                to ROOTFINDER. This produces a scatter plot of all roots
58 %                in the search region.
59 %
60 %   "'quiet'":
61 %
62 %   OUTPUTS
```

```
63 %
64 %   "r":         Vector containing the approximate (complex)
   roots of "f".
65 %
66 %   "WN":        Optional output giving the winding number of
   the region.
67 %
68 %
69 %   Example:
70 %       syms z;
71 %       f = symfun(z^2 + 2, z);
72 %       xsides = [-1, 1];
73 %       ysides = [-2i, 2i];
74 %
75 %       r = rootfinder(f,xsides,ysides,'plot');
76 %
77 %
78 %   Note that ROOTFINDER requires the Symbolic Math Toolbox.
79 %
80 %   Developed by Louis Runcieman (louisruncieman@gmail.com)
   adapted from
81 %   code written by David Franklin (davechrisfranklin@hotmail.
   co.uk).
82 %
83 %   See also METHODCHOICE, QUADSEC, ARGCHANGE, POLYSOLVER,
   QUADGK, SOLVE,
84 %            SYMS, SYMFUN.
85
86 function [r, varargout] = rootfinder(f, xsides, ysides,
   varargin)
```

```
87
88  global quiet
89  method = 0;
90  numchords = []; %Default numchords as empty to satisfy
       variable inputs for subfunctions of rootfinder.
91  plotting = [];
92  %%%%% CHECKING ALL INPUTS CORRECTLY SPECIFIED.
93  if nargin > 7 %Checks correct number of inputs.
94      %Display messaged to inform user of error and to break out
           of program.
95      error('MATLAB:Invalid_inputs', 'Too many inputs to
           rootfinder.')
96  end
97
98  %if ~isa(f,'symfun') %Checks if f is specified as a symbolic
       function.
99  %     %Display messaged to inform user of error and to break
       out of program.
100 %     error('MATLAB:Invalid_inputs', ['Please specify "f" by
       symfun. Eg:\n',...
101 %           'syms z\n', 'f = symfun(z^2 + 1, z)\n', 'rootfinder
       (f,xsides,ysides,method)'])
102 %end
103
104 if ~isreal(xsides) || ~isnumeric(xsides) || length(xsides) ~=
       2 %Check xsides is numeric, real and length 2.
105     %Display messaged to inform user of error and to break out
           of program.
106     error('MATLAB:Invalid_inputs', ['Input "xsides" must be a
           length 2 vector with real entries:\n'...
```

```
107             'rootfinder(f,xsides,ysides,method)'])
108 end
109
110 if real(ysides(1)) ~= 0 || real(ysides(2)) ~= 0 || ~isnumeric(
    ysides) || length(ysides) ~= 2 %Check ysides is numeric,
    purely imaginary and length 2.
111     %Display messaged to inform user of error and to break out
        of program.
112     error('MATLAB:Invalid_inputs', ['Input "ysides" must be a
        length 2 vector with purely imaginary entries:\n'...
113             'rootfinder(f,xsides,ysides,method)'])
114 end
115
116 if xsides(1) >= xsides(2) || imag(ysides(1)) >= imag(ysides(2)
    ) %Checks if region nonempty.
117     %Display messaged to inform user of error and to break out
        of program.
118     error('MATLAB:Invalid_inputs', ['Empty search region:\n'
        ...
119             'rootfinder(f,xsides,ysides,method)'])
120 end
121
122 if ~isempty(varargin)
123     if nargin == 4 %One optional input.
124         if strcmpi(varargin{1}, 'plot')
125             plotting = 1;
126         elseif strcmpi(varargin{1}, 'quiet')
127             quiet = 1;
128         else
```

```
129             error('MATLAB:Invalid_inputs', ['Invalid optional
                    inputs to rootfinder:\n',...
130                 'rootfinder(f, xsides, ysides, ''chords'',
                        numchords, ''plot'')']);
131         end
132     elseif nargin == 5 %Two optional inputs specified.
133         if strcmpi(varargin{1}, 'chords')
134             if isnumeric(varargin{2}) && varargin{2} > 0 &&
                    rem(varargin{2},1) == 0 %Check numchords entry
                    is nonempty and is positive integer.
135                 disp('Applying ''chords'' method...')
136                 method = 1;
137                 numchords = varargin{2};
138                 err = 0;
139             else
140                 error('MATLAB:Invalid_inputs', ['When choosing
                        ''chords'' method, must have positive
                        integer "numchords" specified:\n',...
141                     'rootfinder(f,xsides,ysides,''chords'',
                            numchords).']);
142             end
143         elseif strcmpi(varargin{1}, 'plot') && strcmpi(
                varargin{2}, 'quiet')
144             plotting = 1;
145             quiet = 1;
146         else
147             error('MATLAB:Invalid_inputs', ['Invalid optional
                    inputs to rootfinder:\n',...
148                 'rootfinder(f, xsides, ysides, ''chords'',
                        numchords, ''plot'')']);
```

```
149         end
150     elseif nargin == 6; %3 optional inputs.
151         if strcmpi(varargin{1}, 'chords') && isnumeric(
               varargin{2}) && varargin{2} > 0 && rem(varargin
               {2},1) == 0 %Check numchords entry is nonempty and
               is positive integer.
152             method = 1;
153             numchords = varargin{2};
154             err = 0;
155             if strcmpi(varargin{3}, 'plot')
156                 disp('Applying ''chords'' method...')
157                 plotting = 1;
158             elseif strcmpi(varargin{3}, 'quiet')
159                 quiet = 1;
160             else
161                error('MATLAB:Invalid_inputs', ['Invalid
                      optional inputs to rootfinder:\n',...
162                          'rootfinder(f, xsides, ysides, ''chords'
                             ', numchords, ''plot'')']);
163             end
164         else
165           error('MATLAB:Invalid_inputs', ['Invalid optional
                 inputs to rootfinder:\n',...
166                      'rootfinder(f, xsides, ysides, ''chords'',
                         numchords, ''plot'')']);
167         end
168     elseif nargin == 7; %4 optional inputs.
169         if strcmpi(varargin{1}, 'chords') && isnumeric(
               varargin{2}) && varargin{2} > 0 && rem(varargin
               {2},1) == 0 && strcmpi(varargin{3}, 'plot') &&
```

```
            strcmpi(varargin{4}, 'quiet')
170             method = 1;
171             numchords = varargin{2};
172             err = 0;
173             plotting = 1;
174             quiet = 1;
175         else
176             error('MATLAB:Invalid_inputs', ['Invalid optional
                   inputs to rootfinder:\n',...
177                    'rootfinder(f, xsides, ysides, ''chords'',
                          numchords, ''plot'')']);
178         end
179     else error('MATLAB:Invalid_inputs', ['Too many optional
           inputs to rootfinder:\n',...
180                 'rootfinder(f, xsides, ysides, ''chords'',
                       numchords, ''plot'')']);
181     end
182 end
183 %%%%% INPUT CHECK COMPLETE.
184
185 if double(f == 0) && xsides(1) < 0 && 0 < xsides(2) && imag(
       ysides(1)) < 0 && 0 < imag(ysides(2))
186     r = 0;
187     varargout{1} = 1;
188     return
189 end
190
191 [WN, err] = methodchoice(f, xsides, ysides, method, numchords)
       ; %Find winding number.
192
```

```
193 %Check to see if a root lies on the edge of the region. If so,
        increase
194 %region by 1% in every direction.
195 while abs(real(WN) - round(real(WN))) > 0.0001 || abs(imag(WN)
       ) > 0.0001 || isnan(imag(WN)) || err > 0
196
197     %Increase the size of the region by 1%.
198     xsides = [xsides(1) - 1  / 100 * abs(xsides(1) - xsides(2)
           ), xsides(2) + 1  / 100 * abs(xsides(1) - xsides(2))];
199     ysides = [ysides(1) - 1i / 100 * abs(ysides(1) - ysides(2)
           ), ysides(2) + 1i / 100 * abs(ysides(1) - ysides(2))];
200
201     [WN, err] = methodchoice(f, xsides, ysides, method,
           numchords); %Recalculate winding number.
202
203     %Display message informing user of boundary change.
204     if isempty(quiet)
205         disp('A root near boundary: region has been increased
               by 1%.')
206     end
207 end
208
209
210 if WN >= 0.9999 && WN <= 4.00001 %Check if winding number is
       between 1 and 4.
211
212     WI = zeros(1,round(WN)); %Initialise winding integral
           values.
213
214     for n = 1:round(WN) %Calculate winding integrals.
```

```
215         WI(n) = methodchoice(f, xsides, ysides, 2, n); %
               Applies quadgk algorithm to find winding integral.
216     end
217
218     r = double(polysolver(WI, WN)); %Solve resulting
           polynomials.
219
220 elseif WN > 4.00001 %If winding number greater than 4, apply
    quadsectioning.
221     if isempty(quiet)
222         disp('Quadsectioning region...')
223     end
224     r = zeros(1,round(WN)); %Initialise vector containing
           roots.
225     rn = 1; %Define the initial root number.
226     quadnum = 1; %Initialise quadnum, counts number of
           quadsections.
227     r = quadsec(f, xsides, ysides, r, rn, quadnum, method,
           numchords); %Apply quadsection.
228
229 elseif abs(WN) < 0.00001 %Zero winding number, therefore no
    roots in region.
230     if isempty(quiet)
231         disp('No roots found in the specified region.')
232     end
233     r = [];
234     varargout{1} = 0;
235     return
236 else
237     error('Negative winding number.')
```

```
238 end
239
240 %HHH need to comment these two out when using TestFind.
241 [~, idx] = sort(real(r)); %Sort roots by real part increasing.
242 r = r(idx);
243
244 varargout{1} = round(WN); %Optionally outputs the winding
       number (which is rounded).
245
246
247
248 %Plotting capability.
249 if plotting == 1 %If specified give plot of roots in region.
250     if isempty(quiet)
251         disp('Plotting...')
252     end
253     nroots = size(r, 2);
254     rootcolour = hsv(nroots); %Produce colour map. Each root
          is given a unique colour.
255     figure
256     rectangle('position', [xsides(1), imag(ysides(1)), xsides
          (2) - xsides(1), imag(ysides(2) - ysides(1))]); %Search
           region (includes any size increases).
257     hold on
258     for i=1:nroots
259         plot(real(r(i)), imag(r(i)), 'Color', rootcolour(i,:),
              'Marker', 'x'); %Plot each root.
260     end
261     xlabel('Real axis','Interpreter','LaTex'); %Label for x
          axis.
```

```
262     ylabel('Imaginary axis','Interpreter','LaTex'); %Label for
            y axis.
263     TeXstring = texlabel(f); %Label in LaTeX format.
264     title(['Roots of $f(z) = $' , ' ', TeXstring],'Interpreter
           ','LaTex'); %Title.
265     hold off
266 end
267
268 quiet = [];
269 end %End rootfinder.
```

ARGCHANGE

```
1  %ARGCHANGE
2  %   ARGCHANGE is a subfunction of ROOTFINDER and is not
       intended to be
3  %   used independently. Please see ROOTFINDER.
4  %
5  %   ARGCHANGE determines the contribution from a single edge
       to the winding
6  %   number of a complex function "f" around a contour in the
       complex region
7  %   specified by "xsides" and "ysides" (see ROOTFINDER: INPUTS
       : "f",
8  %   "xsides", "ysides"). This is done by dividing a contour
       from
9  %   "startpoint" to "endpoint" into "numchords" chords. The
       change in
10 %   argument of "f" is then tracked over each of these chords.
        The winding
```

```
11 %   number is increased (or reduced) by 1 if this change is
   greater than
12 %   (or less than) pi. The winding number remains the same
   otherwise. If
13 %   "f" is close to 0 on any chord, "err" is set to 1; for
   more
14 %   information, see METHODCHOICE: OUTPUTS: "err".
15 %
16 %
17 %   [WN, err] = ARGCHANGE(f, startpoint, endpoint, numchords,
   direction)
18 %
19 %   INPUTS
20 %
21 %   "f":          A symbolic function of a single symbolic
   parameter.
22 %
23 %   "startpoint": The starting point of the contour which the
   argument of
24 %                 "f" is to be tracked over.
25 %
26 %   "endpoint":   The end point of the contour which the
   argument of "f" is
27 %                 to be tracked over.
28 %
29 %   "numchords":  The number of chords the contour is to be
   divided into.
30 %
31 %   "direction":  Takes value 0 if the contour is increasing
   along the real
```

```
32 %                 axis or value 1 if increasing along the
   imaginary axis.
33 %
34 %   OUTPUTS
35 %
36 %   "WN":         The winding number contribution from the
   edge starting at
37 %                 "startpoint" and ending at "endpoint".
38 %
39 %   "err:         If nonzero this means a root is located near
    the search
40 %                 contour. If the region has not been
   quadsectioned,
41 %                 ROOTFINDER increases the region size by 1%
   in every
42 %                 direction. If the region has been
   quadsectioned and a
43 %                 root lies near an inner edge then the centre
    point of the
44 %                 quadsection is relocated by ROOTFINDER
   subfunction
45 %                 QUADSEC.
46 %
47 %   See also ROOTFINDER, METHODCHOICE, QUADSEC.
48
49 function [WN, err] = argchange(f, startpoint, endpoint,
   numchords, direction)
50
51 argf = angle(f);
52 stepsize = (endpoint - startpoint) / numchords;
```

```
53
54 %Intialize the winding number.
55 WN = 0;
56
57 %Check the direction of the line and create the appropriate
      vectors.
58 if direction == 0
59 %Create the real and complex points.
60 a = real(startpoint);
61 b = [imag(startpoint) : imag(stepsize) : imag(endpoint)];
62
63 elseif direction == 1
64 %Create the real and complex points.
65 a = [real(startpoint) : real(stepsize) : real(endpoint)];
66 b = imag(endpoint);
67 end
68
69 %Create a complex vector containing the begining/end points of
       the contours
70 z = double(complex(a,b));
71 %Initialize the error variable
72 err = 0;
73 for n = 2 : numchords + 1
74     %Detect if f(z) is close to 0, if so return an error.
75     if abs(double(f(z(n - 1)))) < 0.01
76         %set the error to one
77         err = 1;
78         %Break the function.
79         return
80     %Check to see if arg(f) has crossed the x axis from above.
```

```
81      elseif double(argf(z(n)) - argf(z(n - 1))) <= - pi
82          WN = WN + 1; %Increase the winding number by 1.
83
84      %Check to see if arg(f) has crossed the x axis from above.
85      elseif pi < double(argf(z(n)) - argf(z(n - 1)))
86
87      WN = WN - 1; %Decrease the number winding number by 1.
88      end
89  end
90  end
```

QUADSEC

```
1   %QUADSEC
2   %   QUADSEC is a subfunction of ROOTFINDER and is not intended
        to be
3   %   used independently. Please see ROOTFINDER.
4   %
5   %   QUADSEC quadsections a complex region specified by "xsides
        " and
6   %   "ysides" into 4 subregions for the case when the region
        has a winding
7   %   number greater than 4. QUADSEC then calculates the winding
         number of
8   %   each subregion by ROOTFINDER subfunction METHODCHOICE; if
        a subregion
9   %   has a winding number greater than 4 it is recursively
        quadsectioned. If
10  %   a root is found near the inner edges of the quadsection
        then the centre
```

```
11 %   of the quadsection is relocated.
12 %   If a subregion has a winding number less than or equal to
   4 then it
13 %   calculates and outputs the roots of a function "f" by
   ROOTFINDER
14 %   subfunction POLYSOLVER.
15 %
16 %
17 %   [r, rn] = QUADSEC(f, xsides, ysides, r, rn, quadnum,
   method, numchords)
18 %
19 %   INPUTS
20 %
21 %   "f":          A symbolic function of a single symbolic
   parameter.
22 %
23 %   "xsides":     A length 2 vector giving the search region's
   real limits.
24 %
25 %   "ysides":     A length 2 vector giving the search region's
   imaginary
26 %                 limits.
27 %
28 %   "r":          This is required as an input when QUADSEC
   must be applied
29 %                 more than once. Please see OUTPUTS.
30 %
31 %   "rn":         This is required as an input when QUADSEC
   must be applied
32 %                 more than once. Please see OUTPUTS.
```

```
33 %
34 %   "quadnum":   The count of quadsections applied.
35 %
36 %   "method":    Please see METHODCHOICE: INPUTS: "method".
37 %
38 %   "numchords": Should method 1 or 4 be chosen this positive
   integer
39 %                specifies the number of chords each edge of
   the region is
40 %                to be divided into.
41 %
42 %   OUTPUTS
43 %
44 %   "r":         Vector containing all roots of "f".
45 %
46 %   "rn":        The current number of roots, used to assign
   roots to "r".
47 %
48 %
49 %   See also ROOTFINDER, METHODCHOICE, POLYSOLVER.
50
51 function [r, rn] = quadsec(f, xsides, ysides, r, rn, quadnum,
   method, varargin)
52
53 global quiet
54 if ~isempty(varargin) %Error check for correct input is in
   rootfinder. This is essentially 'protected' class.
55     numchords = varargin{1};
56 end
57
```

```
58 maxquadsec = 20; %Define the maximum number of quadsections. %
      HHH: SHOULD BE DEPEDENT ON SIZE OF REGION?
59
60
61 %Define z to be a symbolic variable.
62 syms z
63
64
65
66 %Find the centre coordinate for the quadsection.
67 cent = [(xsides(1) + xsides(2)) / 2, (ysides(1) + ysides(2)) /
       2];
68
69 %Predefine the err value, used for dectecting boundary roots
      when finding
70 %the winding number by using chords.
71 err = 0;
72
73 %Check whether the maximum number of quadsections has been
      reached.
74 if quadnum >= maxquadsec
75     %Calculate the WN of the region.
76     %WN=WindNum(df_f,xsides,ysides);
77     [WN, err] = methodchoice(f, xsides, ysides, method,
          numchords);
78     %Output error messsage.
79     sprintf('The default maximum number of quadsections has
          been reached. \nThe region found is [[%f,%f],[%fi, %fi
          ]] and has a winding number of %d '...
```

```
80      ,real(xsides(1)),real(xsides(2)),imag(ysides(1)),imag(
            ysides(2)),WN)
81      %Give user choice of options.
82      prompt = ('To continue quadsecting press q.\nTo apply the
            Newton-Raphson Method to the centre of the region press
             nr. \nTo assign the centre as the root press c.\n');
83      str = input(prompt,'s');
84
85       if strcmp(str,'q')
86          %Ask the user how many more quadsections they wish to
                perform.
87          prompt = ('How many more quadsections to you wish to
                perform?\n');
88          %Ask user for numerical input
89          incquad  = input(prompt);
90          %Increase the maxiumum number of quadsections by the
                user defined
91          %amount.
92          maxquadsec = maxquadsec+incquad;
93      elseif strcmp(str,'nr')
94          syms z;
95          df = diff(f,z);
96          %Define the initial guess as the centre point of the
                region.
97          guess0 = cent(1) + cent(2);
98          %Apply the Newton-Raphson method.
99          guess1 = guess0 - f(guess0) / df(guess0);
100         while abs(guess1 - guess0) > 0.0000001
101             guess1 = guess0 - f(guess0) ./ df(guess0);
102             guess0 = guess1;
```

```
103         end
104         %Assign the found roots to r.
105         r(rn : rn + round(WN) - 1) = double(guess1);
106         %Break the function.
107         return
108     elseif strcmp(str,'c')
109         %Set the centre point as the root
110         r(rn : rn + round(WN) - 1) = cent(1) + cent(2);
111         %Increase the root number.
112         rn = rn + WN;
113         %Break the function.
114         return
115      end
116 end
117
118
119 if method == 0
120     %WN=WNContourPA(f,xsides,ysides,cent);
121     WN = methodchoice(f, xsides, ysides, 3, cent);
122 elseif method == 1
123      %[WN,err]=WNChordContour(f,xsides,ysides,cent, numchords)
         ;
124     [WN, err] = methodchoice(f, xsides, ysides, 4, numchords,
        cent);
125 end
126
127 %Intialize the number of times the centre has been moved
128
129 centmov = 0;
130
```

```
131 %Check whether the winding numbers found are integers, if not
    the centre
132 %of the rectangle is moved.
133 while isnan(imag(WN(1))) || isnan(imag(WN(4))) || abs(real(WN
    (1)) - round(real(WN(1)))) > 0.0001...
134         || abs(imag(WN(1))) > 0.000001 || abs(WN(4) - round(WN
               (4))) > 0.0001 || abs(imag(WN(4))) > 0.000001 ||
               err > 0
135
136     %Move the centre point of the region.
137     cent = [cent(1) + abs(xsides(2)-xsides(1)) / (100), cent
           (2) - abs(ysides(2)-ysides(1)) / (100) * 1i];
138     %if the centre has been moved ten times, break the
           function
139
140     if centmov == 10
141         %Output a message to the user informing them of this.
142         sprintf('The winding numbers for the subregions of [[%
               f,%f],[%fi, %fi]] could not be determined.',real(
               xsides(1)),real(xsides(2)),imag(ysides(1)),imag(
               ysides(2)))
143         %break the function.
144         return
145     end
146
147     if method == 0
148         %WN=WNContourPA(f,xsides,ysides,cent);
149         WN = methodchoice(f, xsides, ysides, 3, cent);
150     elseif method == 1
```

```
151         %[WN,err]=WNChordContour(f,xsides,ysides,cent,
                numchords);
152         [WN,err] = methodchoice(f, xsides, ysides, 4,
                numchords, cent);
153     end
154
155     %Display a message informing the user that the centre
            point has been moved.
156     if isempty(quiet)
157         disp('A root has been detected on an inner boundary of
                the quadsection. The centre point for the inner
                boundaries has been moved.')
158     end
159     %increase the number of times the centre has been moved by
            one
160     centmov = centmov+1;
161 end
162
163 %Define the regions in vector form (used for finding the
        winding integrals,
164 %and in the quadsection function).
165 Rx(1, :) = [xsides(1) cent(1)];
166 Ry(1, :) = [cent(2) ysides(2)];
167
168 Rx(2, :) = [cent(1) xsides(2)];
169 Ry(2, :) = [cent(2) ysides(2)];
170
171 Rx(3, :) = [xsides(1) cent(1)];
172 Ry(3, :) = [ysides(1) cent(2)];
173
```

```
174 Rx(4, :) = [cent(1) xsides(2)];
175 Ry(4, :) = [ysides(1) cent(2)];
176
177 for n = 1 : 4
178     %Check the winding number in the subregions.
179     if WN(n) > 4.1
180     %Increase the quadsection number by 1.
181     %disp('Applying another quadsection');
182     quadnum = quadnum + 1;
183     %Apply the quadsection method.
184     [r, rn] = quadsec(f, Rx(n,:), Ry(n,:), r, rn, quadnum,
            method, numchords);
185
186     %If there is between 1 and 4 roots, find the location of
            the roots in
187     %the subregion.
188     elseif WN(n) >= 0.9999 && WN(n) <= 4.00001
189         %Predefine the winding integrals
190         WI = zeros(1,4);
191         %Calculate the winding integrals of R
192         for nu = 1 : round(WN(n))
193             %WI(nu)=WindNum(z^nu*df_f,Rx(n,:),Ry(n,:));
194             WI(nu) = methodchoice(f, Rx(n,:), Ry(n,:), 2, nu);
195         end
196         %Solve the polynomial equation created by the winding
                integrals.
197         r(rn : rn + round(real(WN(n))) - 1)=double(polysolver(
                WI, WN(n)));
198         %Increase the integer used to store the roots.
199         rn = rn + round(WN(n));
```

```
200     end
201 end
202
203 end
```

POLYSOLVER

```
1  %POLYSOLVER
2  %   POLYSOLVER is a subfunction of ROOTFINDER and is not
       intended to be
3  %   used independently. Please see ROOTFINDER.
4  %
5  %   POLYSOLVER applys the MATLAB function SOLVE to polynomials
       of degree
6  %   "WN" = 2, 3, or 4. These polynomials are written in a form
       found from
7  %   the contour integral "z^n * f' / f" where "z" is the
       symbolic parameter
8  %   of "f" (see ROOTFINDER: INPUTS: "f") and "n" is the number
       of roots in
9  %   a given complex region. This contour integral is
       calculated  directly
10 %   using ROOTFINDER subfunction METHODCHOICE: "method" = 2,
       and is the
11 %   input value "WI".
12 %
13 %
14 %   r = POLYSOLVER(WI, WN)
15 %
16 %   INPUTS
```

```
17 %
18 %   "WI": Value of contour integral of "z^n * f' / f" (where "
   z" is the
19 %         symbolic parameter of "f" - see ROOTFINDER: INPUTS:
   "f")
20 %         calculated by ROOTFINDER subfunction METHODCHOICE: "
   method" = 2.
21 %
22 %   "WN": Winding number of the complex region given by "
   xsides" and
23 %         "ysides" - see ROOTFINDER: INPUTS: "xsides", "ysides
   ".
24 %
25 %   OUTPUTS
26 %
27 %   "r":  Vector containing all roots of function "f" in a
   complex region
28 %         given by "xsides" and "ysides" - see ROOTFINDER:
   INPUTS: "f",
29 %         "xsides", "ysides".
30 %
31 %   See also ROOTFINDER, METHODCHOICE, SOLVE.
32
33 function r = polysolver(WI, WN)
34
35 syms z %Define z as symbolic variable.
36
37 if ( 0.9999 < WN) && (WN < 1.0001);
38
39     r = WI(1);
```

```
40
41 elseif ( 1.9999 < WN) && (WN < 2.0001);
42
43
44     %Solve the winding integral equation to give the location
       of the roots.
45     r(1 : 2) = solve(z^2 - WI(1) * z + 1 / 2 * (WI(1)^2 - WI
       (2)) == 0);
46
47 elseif ( 2.9999 < WN) && (WN < 3.0001);
48
49     %Solve the winding integral equation to give the location
       of the roots.
50     r(1 : 3) = solve(z^3 - WI(1) * z^2 + 0.5 * (WI(1)^2 - WI
       (2)) * z +...
51                 0.5 * WI(1) * WI(2) - (1 / 6) * WI(1)^3 - (1 /
                      3) * WI(3) == 0);
52
53 elseif ( 3.9999 < WN) && (WN <= 4.0001);
54
55     %Solve the winding integral equation to give the location
       of the roots.
56     r(1 : 4) = solve(z^4 - WI(1) * z^3 + 0.5 * z^2 * (WI(1)^2
       - WI(2)) +...
57                 0.5 * z * (WI(1) * WI(2) - (1 / 3) * WI(1)^3 -
                      (2 / 3) * WI(3)) +...
58                 (1 / 24) * (WI(1)^4 - 6 * WI(1)^2 * WI(2) + 3 *
                       WI(2)^2 +...
59                 8 * WI(1) * WI(3) - 6 * WI(4)), z, 'MaxDegree',
                       4);
```

```
60 end
61 end
```