A parallel algorithm to find the zeros of a complex analytic function.

Michael H. Meylan* Lutz Gross†

(Received 2 February 2002; revised 15 December 2002)

Abstract

Motivated by the general non-linear eigenvalue problem, we present a parallel algorithm to find the zeros of a complex analytic function in a given region. The algorithm is based on a two dimensional version of the bisection algorithm and is implemented in parallel using a master-slaves programming model. The master is responsible for organising the slaves while the slaves are responsible for determining if a given region contains any zeros. The results from the test calculations show that this algorithm achieves good efficiency provided that the number of processors does not exceed four time the number of zeros in the initial region.

*Institute of Information and Mathematical Sciences, Massey University, Private Bag 102-904 NSMC, Auckland, New Zealand.
mailto:M.H.Meylan@massey.ac.nz

†Mathematical and Information Sciences, CSIRO, Private Bag 10, Clayton South, Victoria, Australia

[0]See http://anziamj.austms.org.au/V44/E036 for this article, © Austral. Mathematical Soc. 2003. Published February 20, 2003 ISSN 1446-8735

Contents

1 Introduction

In this paper we present a method to determine the zeros of a complex analytic function in a given bounded region S. The method, which is based on elementary results from complex analysis, is very robust and will find every zero within the given region for a very large class of functions. For this robustness a high computational price must be paid and for this reason we have designed the algo-

rithm to run on a parallel computer.

There are many possible reasons why one might wish to find the zeros of a complex analytic function. However, most zero search algorithms tend to be either more specialised, for example finding the zeros of a polynomial, or more general, for example finding the zeros of an arbitrary system of functions. The motivation for the present work is the most general form of the non-linear eigenvalue problem in which the matrix is an arbitrary value of the eigenvalue parameter.

There are many algorithms to determine the zeros of specific analytic functions. The best known are the iterative methods to find the eigenvalues of a square matrix. The idea of using complex variable theory to find the zeros of a polynomials has been proposed several times [1, 2, e.g.]. However the current method used by programs such as MATLAB is based on rewriting the polynomial problem as a matrix eigenvalue problem. Recent research on the nonlinear eigenvalue problem has concentrated on extending the existing methods to problems of only slightly greater complexity, such as the quadratic eigenvalue problem [3]. However, there are some cases in which the non-linear eigenvalue problem is of full complexity [4] and we are faced with finding the eigenvalues of a matrix which is an arbitrary function of the eigenvalue parameter. By taking the determinant of this matrix we can pose this problem as one of finding the zeros of an analytic function.

The method which we present to find the zeros of a complex analytic function is based the on following result from elementary complex variable theory. The variation of the argument of the analytic function around the boundary of a region is 2π times the number of zeros within the region (since there are no singularities within the region). The algorithm is a simple generalisation of the bisection algorithm for functions of a real variable. At each step of

the iteration the region is subdivided and each subregion is searched. If the subregion contains a zero it is further subdivided and the process is repeated. Furthermore, this algorithm is implemented to run on a parallel computer using a master-slaves programming model.

2 Motivation: The non-linear eigenvalue problem.

The present work is motivated by trying to solve the non-linear eigenvalue problem in its most general form.

The linear eigenvalue problem can be written as the problem of finding the values of λ for which the following complex analytic function

$$f(\lambda) = \det[\mathbf{M} - \lambda\mathbf{I}]$$

is zero ($\mathbf{M}$ is a real or complex square matrix and $\mathbf{I}$ is the identity matrix of the same size as $\mathbf{M}$).

In the non-linear eigenvalue problem the parameter λ appears in a more complicated manner in the matrix. In the most general nonlinear eigenvalue problem we must find the values λ for which

$$f(\lambda) = \det[\mathbf{M}(\lambda)] = 0$$

where the matrix is an arbitrary analytic function of the parameter λ. This kind of problem arises when modelling situations involving complicated interactions, for example between a fluid and a structure. Recently Meylan [4] showed that the modes of vibration of a thin plate on shallow water could be determined by solving such a generalised eigenvalue problem.

State-of-the-art solution methods for solving the linear eigenvalue problem do not refer to the analytic function f directly but transform M into a diagonal or nearly diagonal matrix. These methods cannot be applied to the general non-linear eigenvalue problem; however, generalizations for the quadratic eigenvalue problem exist. So in the general non-linear case one has to refer to the problem of finding the roots of f. Of course this approach could be used for the linear and quadratic case as well but one will expect to end up with a less efficient method.

3 The search algorithm

The major difficulty in determining the zeros of a complex analytic function $f(\lambda)$ is that it is not possible to use a rapid zero search algorithm such as the Secant or Newton method because these methods require an initial starting guess which is close to the unknown zero. The algorithm which we propose is a generalisation of the well-known bisection algorithm to find roots of a real-valued function. It is extremely simple and robust but it requires extensive computational resources.

The idea is as follows: The algorithm starts form an initial square S with boundary ∂S in the complex plane. It is assumed that f is analytic in the square and in a neighbourhood of the square. The search algorithm will try to find all the zeros of $f(\lambda)$ which lie within this square S. In practice we generally seek the zeros with some property, such as those with smallest magnitude, so that the restriction to a bounded domain does not give any special problems.

Assuming that the square contains no zeros on its boundary, we use the result that the number of zeros of $f(\lambda)$, which we denote by N, that lie within the square is given by the variation $\Delta_{\partial S}$ of the

argument of f anti-clockwise along the closed curve ∂S, that is

$$N = \frac{1}{2\pi}\Delta_{\partial S}\left(\arg f\left(\lambda\right)\right) . \tag{1}$$

If the square contains some zeros, which is indicated by $N > 0$, this square is then subdivided into four squares and the search is carried out in each of these squares.

In each step of this iterative process we have to search a set of squares. The squares being found to contain a zero are further subdivided. If a square does not contain a zero it is dropped. In the unlikely event that all squares searched in an iteration step contain at least one zero the number of squares to be considered in the next step is increased by a factor of four. If there is only one single square containing a zero, there are only four squares searched in the next iteration step. A practical situation will lay between these two extreme cases. Figure 1 illustrates the first four iterations of the search method for an example problem with four zeros which are marked with a cross.

3.1 Determining the change in argument

We discuss here the algorithm used to track the change in the argument of f. The change in argument along ∂S is determined by evaluating $f\left(\lambda\right)$ at certain points around the boundary of the square:

We begin in a corner of the square. We evaluate the function $f\left(\lambda\right)$ at this point and determine the argument. We then consider a point a distance d along a side of the square and evaluate the argument at this point. If the change in argument (taking account the flip from $-\pi$ to π) is less than some value θ we accept this new point. If the change in argument is greater than θ then we consider

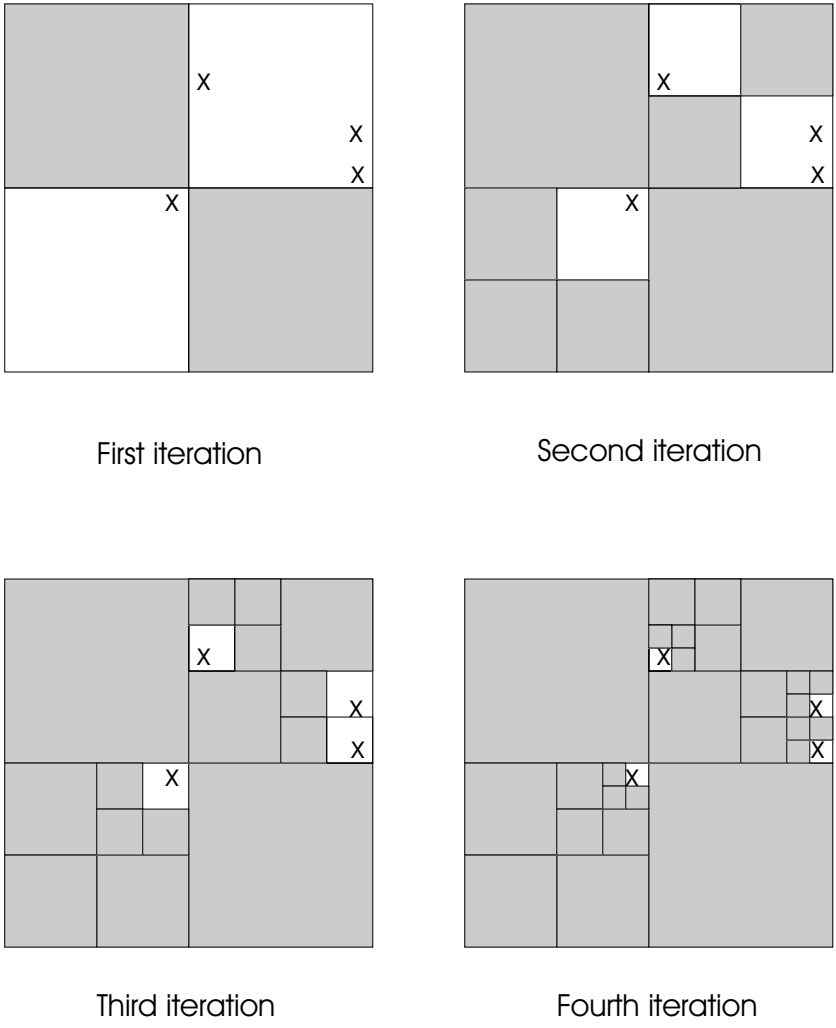

FIGURE 1: Four iterations of the search algorithm for an example problem. The four zeros are marked with a cross and the grey denotes the regions which have been excluded after each iteration.

a new point a distance $d/2$ along the square and test to see if the change in argument is less than θ. The halving is repeated until the change in argument is less than θ. We then repeat the same process starting at the last point we determined the argument. The process stops when the entire boundary has been traversed and the initial point has been reached. The value for change in argument is then set to the change in the argument for the two values at the initial point. Our tests show that the step size d can be set quite large (half the side length of the square for example), while the change in angle should be no greater than $\pi/10$. For these values one achieves a good balance for accuracy with minimising the numerical cost.

The algorithm is likely to fail if we have two or more zeros close together which are close to the boundary. In this case the algorithm can step right past the zeros with out picking up that there are in fact two zeros or more. However, although the value for N returned by the algorithm is smaller than the actual number of roots in S, it still return a positive value which is sufficient to decide that S has to be subdivided.

A principle problem of the algorithm does occur when a zero lays on or close to the boundary of the square. In this case it can happen that a zero is not detected although the square contains a zero. There is a way around this problem at only little extra costs: one can increase the size of the initial square and of each sub-square when the square is divided. This would give some overlap of the sub-squares so that the zeros could not lie close to the boundary of two squares. We suggest that in certain cases this may be the preferred method, especially in situations where clustering of zeros is likely to occur.

Obviously one could construct a function for which we would fail to find a zero within the particular square, for example by choosing the function to have value 1 at every point where we evaluate $f(\lambda)$

around the square and yet to contain a zero within the square. We consider these cases as unlikely to occur in practice. However, one can reduce the risk of a failure of the algorithm by choosing sufficiently small θ and d or, which is more reliable by comparing values for two different choices for θ and d. Nevertheless, this will not guarantee that a root in a square is detected.

3.2 Determining when to use the secant method

Once a square contains one zero only and the square is small enough, it is more efficient to use the secant or Newton method to improve the accuracy of a zero to the desired tolerance. The principle problem is determining when we are close enough to the a zero in order to switch to the secant or Newton method. In practice, we always use the secant method because in a close neighborhood to the zero its performance is close to that of the Newton method but has the advantage that it does not require the evaluation of $f'(\lambda)$. It is reasonable to use the secant method immediately we have determined that only one zero lies within the square since the secant method is numerically cheap. If the secant method fails to converge to a zero within the square, we switch back to the process of subdivision and try the secant method on the sub-square which contains the zero.

As pointed out above, it can happened that the number of zeros is underestimated. Therefore it can happen that the secant or Newton method converges to a zero in a square but there are more zeros in the square. However, continuing the subdivision strategy would likely pick-up these zeros. Here the user has to make decision between reliability and speed.

4 Implementation in parallel

The major aim of our work is to implement the algorithm on a parallel computer. Our reason for this is that the calculation, especially for large dimensional $\mathbf{A}(\lambda)$ is extremely computationally demanding. Furthermore, the algorithm is well suited to parallelisation since the tasks of searching the squares in a particular iteration are independent and can therefore easily be executed in parallel. We have implemented the most straightforward version of this algorithm which works as follows.

We use a master-slaves programming model: one particular process, called the master, organizes the work of the remaining slave processors. The action of the slaves is the most straightforward. Each slave is given the parameter of a square in the complex plane in which it must determine the number of zeros using equation (1). Once it has finalised this task it returns this information and waits to be given a further task.

The master holds the table of the squares to be searched and allocates squares to any slaves that are free. Once a slave has completed its calculation this information is used to update the job table as follows: The processed square is removed from the table and, if the square contained a zero, four new squares are added to the table. After this up-dating step, the master allocates squares waiting to free slaves until either there are no free slaves or no squares waiting to be processed. In the case that there are more slaves than squares to be processed there will be idle slaves. The algorithm terminates when the zeros are determined to some given accuracy.

5 Efficiency

We will discuss the efficiency of our algorithm in the context of the generalised bisection search. We will not consider issues that arise when the algorithm includes the secant method. The major problem with any optimal parallel algorithm is to ensure that the workload is distribute equally across the available processors and that all processors are kept busy. For our algorithm this means that we must ensure that the slaves are working at all times. The most significant idleness arises because the slaves cannot be allocated a job. Typically this situation occurs at the beginning of the iteration process when we are dealing with a small number of squares. However, we will not be able to generate enough jobs if the number of zeros in the initial square is small. Once the iteration process has proceeded through the initial interactions it can be expected that the sub-squares to be processed are small enough to contain just one zero each. As each sub-square will produce four new sub-squares to be searched the number of jobs created for the next iteration step is $4\times$ number of zeros. Therefore, we expect that, in order to keep the slaves busy, the number of jobs should be equal or greater than the number of slaves we are using. This can be written as the following condition:

$$4 \times (\text{number of zeros in } \partial S) \leq (\text{number of slaves}). \tag{2}$$

Notice that we determine the number of zeros in the initial square from equation (1).

In order to keep an arbitrary number of slaves busy one can introduce a subdivision into a suitable number of squares (or more general into rectangles) rather then four squares. The optimal number of squares could be determined by an equation similar to (2) after the first iteration step.

6 Examples

We present here some example calculations. The examples where calculated on a cluster of 16 500 MHz PentiumIII PCs under LINUX. The computers are linked through an 100 Mbit Ethernet switch. The algorithm has been implemented in C++ using the Gnu compiler. In all case the search is terminated once the side length is smaller then $10^{-8}\times$ the modulus of the centre of the square and we do not use the secant method at any point. To slow down the compute time for each slave we determine the argument by evaluating the function at 10,000 points around the square. This is because we do not wish the total compute time to be dominated by communication costs.

6.1 Example: $f(\lambda) = \sin(\lambda)$

We begin by finding the zeros of $f(\lambda) = \sin\lambda$. using various initial squares. The results are shown in Table 1. The columns N_0 and p gives the number of roots in the initial square and the number of slaves which have been used, respectively. The time taken is given by T_p and is in seconds. The value η is the ratio of T_p and the product of the number of slaves and time T_1 when using one slave only: if η is close to 1 we have achieved high efficiency; if η is close to zero the efficiency is very poor. Note that we do not consider the timing for the best possible algorithm for one processor which would be the true efficiency. Also, there is some randomness in these timings which is typical of the communication randomness of parallel algorithms. As explained previously, the process of determining the number of zeros, which each slave must perform, has been slowed by requiring a large number of function evaluations on the boundary of the square. This is to ensure that there is a more typical load

centre	side length	N_0	p	T_p	η
(0,0)	4	1	1	4.41	1
			2	2.26	0.97
			4	1.16	0.95
			8	0.75	0.73
			15	0.71	0.41
(0,0)	8	3	1	10.19	1
			2	5.88	0.87
			4	3.02	0.84
			8	1.42	0.9
			15	1.03	0.66
(0,0)	16	5	1	18.9	1
			2	9.51	0.99
			4	4.88	0.97
			8	2.33	1.01
			15	1.24	1.01

TABLE 1: A table of times, T_p, to find the zeros of $f(\lambda) = \sin(\lambda)$ in the squares with centre and side lengths as shown. N_0 is the number of zeros in the square, p is the number of processors and η is the efficiency.

centre	side length	n, k	p	T_p	η
(0,0)	1.99	5,0	1	30.2	1
			2	15.5	0.97
			4	7.7	0.98
			8	4.04	0.93
			15	2.73	0.74
(0,0)	1.99	5,5	1	26.2	1
			2	13.11	1
			4	6.55	1
			8	3.62	0.9
			15	2.56	0.68
(0,0)	1.99	5,10	1	20.12	1
			2	10.33	0.97
			4	5.14	0.98
			8	2.72	0.92
			15	2.2	0.61

TABLE 2: A table of times, T_p, to find the zeros of a polynomial with clustered zeros in the squares with centre and side lengths as shown. The polynomial is defined by equation (3) and k and n are parameters which define the polynomial. η is the efficiency.

balance for a practical problem, otherwise the communication time would be dominant. The results in Table 1 show that the algorithm retains a high efficiency provided that the number of processors is no greater than four times the number of zeros. Once this threshold is crossed, the efficiency drops significantly. This is exactly as expected from equation (2).

6.2 Example: A polynomial with clustered zeroes

In this example we consider the following polynomial

$$p_k(\lambda) = \left(\lambda - \frac{1}{2^k}\right)\left(\lambda - \frac{1}{2^{k+1}}\right)\cdots\left(\lambda - \frac{1}{2^{k+n}}\right) \tag{3}$$

where k and n are integers. This zeros of this polynomial are clustered and therefore we expect that the algorithm will lose efficiency faster as the number of processors is increased than was the case for the previous example. We consider an initial square has centre 0 and side length 1.99 (to ensure that none of the zeros lie of the boundary on the sub-squares). The number of roots $n = 5$ and $k = 0, 5, 10$. The results are shown in Table 2. See from these results that, as the clustering of zeros increases, there is a marked decrease in the efficiency.

6.3 Example: Increasing the number of sub-squares

One strategy to try and improve the speed when there is a large number of processors available is to increase the number of sub-squares. Table 3 shows the results for the calculation for $f(\lambda) = \sin\lambda$ in the case when there is only one zero. The number of sub-squares is 4, 9, and 16 respectively. Perhaps because of the increased communication cost, the results shown that, even with 15 slaves there is no efficiency gain to be made by increasing the number of sub-squares.

centre	side length	Number of Sub-squares	p	T_p
(0,0)	4	4	1	4.41
			2	2.26
			4	1.16
			8	0.75
			15	0.71
(0,0)	4	9	1	6.16
			2	3.4
			4	1.76
			8	1.01
			15	0.83
(0,0)	4	16	1	9.49
			2	4.89
			4	2.5
			8	1.39
			15	1.00

TABLE 3: A table of times, T_p, to find the zeros of $f(\lambda) = \cos(\lambda)$ in the squares with centre and side length as shown. The number of sub-squares used at each iteration is shown. p is the number of processors.

6.4 Example: Vibrations of a floating thin plate

The final example is taken from [4] in which the zeros of the determinant of the following matrix

$$\mathbf{M}(\lambda) = \begin{bmatrix} \mu_1^4 e^{-\mu_1 b} & \mu_2^4 e^{-\mu_2 b} & \cdots & \mu_6^4 e^{-\mu_6 b} & 0 & 0 \\ \mu_1^5 e^{-\mu_1 b} & \mu_2^5 e^{-\mu_2 b} & \cdots & \mu_6^5 e^{-\mu_6 b} & 0 & 0 \\ \mu_1^4 e^{\mu_1 b} & \mu_2^4 e^{\mu_2 b} & \cdots & \mu_6^4 e^{\mu_6 b} & 0 & 0 \\ \mu_1^5 e^{\mu_1 b} & \mu_2^5 e^{\mu_2 b} & \cdots & \mu_6^5 e^{\mu_6 b} & 0 & 0 \\ e^{-\mu_1 b} & e^{-\mu_2 b} & \cdots & e^{-\mu_6 b} & -e^{-i\lambda b} & 0 \\ \mu_1 e^{-\mu_1 b} & \mu_2 e^{-\mu_2 b} & \cdots & \mu_6 e^{-\mu_6 b} & -i\lambda e^{-i\lambda b} & 0 \\ e^{\mu_1 b} & e^{\mu_2 b} & \cdots & e^{\mu_6 b} & 0 & -e^{-i\lambda b} \\ \mu_1 e^{\mu_1 b} & \mu_2 e^{\mu_2 b} & \cdots & \mu_6 e^{\mu_6 b} & 0 & i\lambda e^{-i\lambda b} \end{bmatrix} \tag{4}$$

give the natural frequencies of the a floating thin plate on shallow water. In equation (4) μ_1 to μ_6 are the roots of the following polynomial

$$\beta\mu^6 + \mu^2 + \lambda^2 = 0\,. \tag{5}$$

Table 4 shows the times taken to determine the zeros of $\det[\mathbf{M}(\lambda)]$ in the square of side length 2 centered at $(1, 1)$. The algorithm shows the good efficiency that would be expected for an initial region with 8 zeros.

7 Summary

Motivated by the most general nonlinear eigenvalue problems we have presented a parallel algorithm to find the zeros of a complex analytic function. This algorithm was based on a simple process of subdivision. The algorithm was implemented in parallel using a master-slaves programming model. As was expected, the results

centre	side length	N_0	p	T_p	η
(0.5,0.5)	1	8	1	168.72	1
			2	84.71	1
			4	43.61	0.97
			8	21.93	0.96
			15	10.29	1.09

TABLE 4: A table of times, T_p, to find the zeros of $\det[\mathbf{M}(\lambda)]$ where $(\mathbf{M}(\lambda)$ is given by equation (4). p is the number of processors and η is the efficiency.

from the test calculations showed that this algorithm achieves good efficiency provided that the number of processors does not exceed four time the number of zeros in the initial region.

Acknowledgement: I appreciated the helpful comments of the reviewer, especially the suggestion of enlarging the search squares to avoid problems with zeros close to the boundary. This research was supported in part by the Marsden Fund.

References

[1] H. S. Wilf, A global bisection algorithm for computing the zeros of polynomials in the complex plane, *J. Assoc. Computing Machinery*, **25(3)**, pp 414–420, 1978, E238

[2] C. Caretensen and M. S. Petkovic, On iterative methods without derivatives for the simultaneous determination of polynomial zeros, *J. Comp. & Appl. Math.*, **45**, pp 251–266, 1993 E238

[3] Z. Bai, Demmel, J. Dongarra, A. Ruhe, H. Van der Vorst, *Templates for the solution of algebraic eigenvalue problems: a practical guide*, SIAM, Philadelphia, 2000. E238

[4] M. H. Meylan, Spectral solution of time dependent shallow water hydroelasticity, *J. Fluid Mechanics,* **454**, pp 387–402, 2002. E238, E239, E252