Postprint of article in *Proceedings of the 26th Annual International Computer Software and Applications Conference* (*COMPSAC 2002*), IEEE Computer Society, Los Alamitos, CA, pp. 327–333 (2002)

Metamorphic Testing of Programs on Partial Differential Equations: a Case Study [*] [†]

T.Y. Chen
Swinburne University of Technology
Melbourne, Australia

Jianqiang Feng *and* T.H. Tse [‡]
The University of Hong Kong
Pokfulam, Hong Kong

Abstract

We study the effect of applying metamorphic testing to alleviate the oracle problem for numerical programs. We discuss a case study on the testing of a program that solves an elliptic partial differential equation with Dirichlet boundary conditions. We identify a metamorphic relation for the equation and demonstrate the effectiveness of metamorphic testing in identifying the error. The relation identified in the paper should also be applicable to other numerical methods that yield better approximations on the refinement of grid points or step sizes.

Keywords: Program testing, metamorphic testing, oracle problem, partial differential equations

1. Introduction

Numerical programs are vital to our daily lives. They have been used not only in various theoretical disciplines, but also engineering and medical practices including mission-critical and safety-critical applications. Unfortunately, despite the importance of the quality of numerical packages, we are far from doing a good job [17].

Like other testing contexts, we usually assume that we can verify the actual outputs of numerical software against some expected results. We call the mechanism of checking the correctness of the test output as a *test oracle* [5, 15]. Developers of numerical software generally adopt the following mechanisms as test oracles:

(*a*) Comparing with analytical solutions, simulation results, tabulated values, or hand-calculations [15].

(*b*) Verifying with standard mathematical libraries or reference software packages [7].

Test oracles, however, may not be available in every program. This is the so-called *oracle problem*. It is especially the case for numerical software. Because of truncation errors (due to truncating an infinite series into a finite series), rounding errors (due to the digital representation of floating-point numbers) and the propagation of errors in the computing process, numerical computation introduces unwarranted errors that will affect the final results. Thus, we cannot find exact solutions to numerical problems. Instead, we can aim at bounding the errors of numerical solutions so that they satisfy given precision requirements. With the efforts of mathematicians, by applying special techniques, we can assure the precision of some numerical functions, such as elementary functions [13]. It is, however, very difficult to analyze the errors in complex numerical computation [19, 20].

There are reputable mathematical libraries, such as IMSL [1] and NAG [2], which have matured through intensive testing and real-life applications. In fact, some popular numerical libraries have been refined gradually because of errors identified throughout the operational lives of the programs. We can compare some of the results of our numerical software with these libraries or similar reference software, but how do we handle other results that may

[†] This work is supported in part by grants of the Research Grants Council of Hong Kong (Project Nos. CityU 1118/99E and HKU 7029/01E) and a research and conference grant of The University of Hong Kong.

[‡] All correspondence should be addressed to Prof. T.H. Tse, Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong. Email: "thtse@cs.hku.hk".

involve special features not available in standard libraries? By the same argument, tabulated values and analytical solutions may not be available in every application.

In this paper, we would like to study the effect of applying metamorphic testing [8, 10, 11] to alleviate the oracle problem for numerical programs. We shall discuss a case study on the testing of a program that solves an elliptic partial differential equation with Dirichlet boundary conditions. This type of partial differential equation is useful in many practical applications such as the vibrations of rods and beams, the motion of fluid waves, and the transmission of sound and electrical signals.

The rest of the paper is organized as follows: Section 2 gives a brief review of the testing of numerical software in the absence of an oracle. It is followed by the main section. We look into a numerical method that solves an elliptic partial differential equation with Dirichlet boundary conditions. We examine a program that implements the method, inject a fault into the program, and try whether special case testing can reveal the fault. Then we identify a metamorphic relation for the equation and study the effectiveness of metamorphic testing in identifying the error. Section 4 concludes the paper.

2. Testing of Numerical Programs in the Absence of an Oracle

Three approaches have been proposed [16] to provide an alternative oracle for numerical programs when a straightforward test oracle cannot be found. First, we can use experimental results to check the programs. However, even if we can detect some differences between the numerical solutions and experimental results, it is difficult to verify whether they are due to program bugs or errors in the modeling of the physical phenomena. Secondly, we can compare the results generated by different numerical programs, but then it is difficult to differentiate between the correct software and the faulty one. Thirdly, we can run the software using different parameter settings and verify the consistency of the results. However, different environmental settings may actually produce different results.

Instead of trying to find an alternate oracle, an extensively used technique for testing numerical programs is to verify whether they satisfy selected identity relations of the functions being implemented. For example, given a program that computes the sine function, we can verify whether the implementation exhibits the relation $\sin^2 x + \sin^2 (\pi/2 - x) = 1$. This technique has been explained, for instance, in Cody and Waite [13] and Weyuker [21].

Data diversity is a related technique developed by Ammann and Knight [4]. It is targeted to overcome the problems associated with *N*-version programming, and has therefore been developed from the perspective of fault tolerance rather than fault detection. As a consequence, properties used in data diversity are intrinsically limited to identity relations.

Another technique of *program checkers* was developed by Adleman et al. [3] and Blum and Kannan [6]. It checks whether an implemented program satisfies the specified function with a probability level defined by the user.

More recently, a testing method called *metamorphic testing* was put forward by Chen et al. [8, 10]. The underlying philosophy is that, even though a successful test case does not reveal any failure, it may still provide testers with useful information. Follow-up test cases can be used to verify certain necessary properties of the program, no matter whether a testing oracle exists or not. If the necessary properties do not hold, the program must be incorrect. Thus, metamorphic testing renders a new approach based on a series of test cases and necessary properties of the program under test. In [11], Chen et al. further illustrated the usefulness of metamorphic testing by integrating it with fault-based testing.

Let $f : X -> Y$ be a function with domain X and co-domain Y. Suppose f satisfies some property R_f that can be expressed as a relation between a series of elements x_1, x_2, ..., $x_n \in X$, where $n > 1$, and the corresponding series of elements $f(x_1)$, $f(x_2)$, ..., $f(x_n) \in Y$. This relation R_f will be called a *metamorphic relation* of f. Take the tangent function as an example. For any two inputs x_1 and x_2 such that $x_2 = \pi + x_1$, we must have $\tan\ x_2 = \tan\ x_1$. We can express this as a metamorphic relation:

$$R_{\tan}:\ x_2 = \pi + x_1 \ \rightarrow\ \tan\ x_2 = \tan\ x_1.$$

Any program that implements f must satisfy all the metamorphic relations R_f defined for f. They are necessary conditions for the correctness of the program. For instance, to verify a metamorphic relation $R_{\tan}$ above, we need two executions. The first input is any real number x_1 and the second is the number $x_2 = \pi + x_1$.

There are several differences between metamorphic testing and other methods that propose to test programs against selected properties of implemented functions.

(*a*) Metamorphic properties are not only identity relations but can also be expressed in other forms such as inequalities. This paper exactly describes an application of inequality relations to the testing of partial differential equations. Such inequalities are not covered by the other techniques, except a couple of error bound examples in [21].

(*b*) Data diversity is a fault-tolerance technique designed to process the original test cases in alternate ways using the same program. Hence, only the re-expressed forms of the original test cases will be considered.

On the other hand, metamorphic testing is a fault-detection technique. Even though follow-up test cases are employed, they are not limited to re-expressed forms of the original test cases.

(*c*) Program checkers aim at providing a probabilistic oracle, in order to estimate whether the corresponding output is likely to be correct. The generation of additional test cases is only a by-product of the system. Metamorphic testing, however, is designed as a property-based test case selection strategy rather than for providing some form of alternative oracle.

Readers may also refer to [12] for more discussions.

We note that it will not be necessary to have a test oracle when applying metamorphic testing. This can help alleviate the oracle problem in many situations, such as the partial differential equations that we are about to discuss.

3. Testing of Programs on Partial Differential Equations

A great deal of effort and mathematical sophistication were required to solve partial differential equations analytically in a limited number of applications. The introduction of digital computers facilitated us to solve these equations by means of numerical computation [9]. Hence, it attracted much research into the numerical methods for generating solutions more effectively and efficiently. How do we test the programs that implement these numerical methods? In the absence of analytical and tabulated solutions, programs on partial differential equations are also subject to the oracle problem as described in Section 1.

We shall focus our discussion on a practical problem in thermodynamics, namely the distribution of temperatures on a square plate. We assume that the plate is insulated, that is, there will be no heat transfer to or from the environment. The temperatures at the boundary of the plate are given. Along each edge of the plate, the temperature is homogeneous. Given enough time, the heat potential of the plate will reach stability. We wish to find the temperature at each point on the plate. The problem can be modeled by a Laplace equation

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0.$$

with Dirichlet boundary conditions.

Laplace equations can be used not only to represent this case, but also to model many practical applications such as fluid flows as well as gravitational and electrostatic potentials [18]. The testing technique described in this paper is applicable as long as the numerical method for solving the equation converges.

In order to calculate the temperature T at any point P, central-difference technique is used to approximate the second derivatives:

$$\frac{\partial^2 T}{\partial x^2}(P) \approx \frac{T(P_L) - 2T(P) + T(P_R)}{h^2}$$
$$\frac{\partial^2 T}{\partial y^2}(P) \approx \frac{T(P_A) - 2T(P) + T(P_B)}{h^2}$$

where P_A and P_B are the points at a fixed distance h above and below P, and P_L and P_R are the points at the same distance h to the left and right of P.

The plate will be divided uniformly in the form of mesh grids, namely

G_1: 3×3 mesh grid,
G_2: 7×7 mesh grid,
G_3: 15×15 mesh grid,
G_4: 31×31 mesh grid,
G_5: 63×63 mesh grid,

and so on. Figure 1 shows G_1 to G_4. The step size for G_i is h_i, such that

$$h_1 = 2h_2 = 4h_3 = 8h_4 = 16h_5.$$

As the densities of the mesh grids increase from G_1 to G_5, the accuracy of the solution will increase and the anticipated error will decrease accordingly [14, 19]. If the step size h is uniform, similarly to the case in Figure 1, the anticipated error is in the order of h^2; otherwise, it is in the order of h [19].

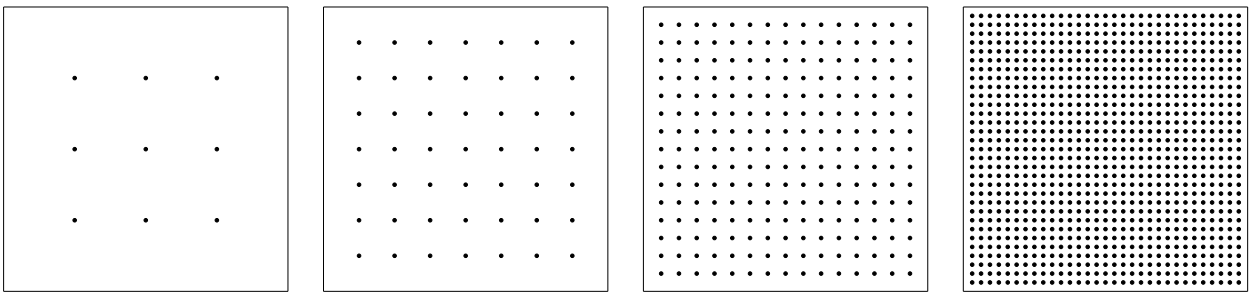
3.1. Sample Program on Partial Differential Equation

The program we test is adapted from [14]. It uses the "alternating direction implicit" method to solve the partial differential equation mentioned above. Given the boundary conditions and a step size, the adapted program can compute the temperatures at the grid points until the changes in temperatures between two consecutive iterations are smaller than the stopping criterion. We shall refer to this program as *PDE*.

In order to illustrate the effectiveness of metamorphic testing, we would like to inject an error into the *PDE* program. The following is a correct statement in the program:

$$\text{if } (fabs(uMat[i][j] - vMat[j][i]) > larg)$$
$$larg = fabs(uMat[i][j] - vMat[j][i]);$$

These variable names were used in the original published program in [14]. The variables *uMat* and *vMat* represent temperature matrices for two consecutive iterations. The



G_1: 3×3 Mesh Grid G_2: 7×7 Mesh Grid G_3: 15×15 Mesh Grid G_4: 31×31 Mesh Grid

Figure 1. Improving Precision by Refining the Mesh Grids

use of $[i][j]$ and $[j][i]$ in these matrices explains the name of the popular "alternating direction implicit" method. If the temperatures at the points are calculated by rows and then by columns in one iteration, they will be calculated by columns and then by rows in the next iteration. Thus, $fabs(uMat[i][j] - vMat[j][i]) = \left|uMat[i][j] - vMat[j][i]\right|$ is the change in temperature between two consecutive iterations at a specific point. The variable *larg* is the maximum change in temperature between two iterations for all points. If the change in temperature $\left|uMat[i][j] - vMat[j][i]\right|$ at a certain point exceeds *larg*, then *larg* should take up this value. Otherwise, *larg* should remain unaffected.

Suppose we inject an error by amending *vMat* to *uMat* in the first line, thus:

$$\begin{aligned}&\text{if } (fabs(uMat[i][j] - \mathbf{uMat}[j][i]) > larg)\\ &\qquad larg = fabs(uMat[i][j] - vMat[j][i]);\end{aligned}$$

This fault is difficult to detect because both the correct and faulty versions of the *PDE* program yield exactly the same results when mesh grids G_1 and G_2 are being tested, and yield a fairly close result when mesh grid G_3 is being tested.

We would like to examine what actually happened. In the faulty program, when we reached the 44th iteration for the mesh grid G_1, we obtained the following. Note that only parts of the results are shown.

(a) $uMat[2][3] = 136.964279$
$uMat[3][2] = 341.607147$
$vMat[3][2] = 136.964309$
$\left|uMat[2][3] - uMat[3][2]\right| = 204.642868 > larg$
$\therefore\ larg = \left|uMat[2][3] - vMat[3][2]\right| = 0.000030$

(b) $uMat[3][1] = 304.285736$
$uMat[1][3] = 75.714302$
$vMat[1][3] = 304.285705$
$\left|uMat[3][1] - uMat[1][3]\right| = 228.571434 > larg$
$\therefore\ larg = \left|uMat[3][1] - vMat[1][3]\right| = 0.000031$

(c) $uMat[3][2] = 341.607147$
$uMat[2][3] = 136.964279$
$vMat[2][3] = 341.607147$
$\left|uMat[3][2] - uMat[2][3]\right| = 204.642868 > larg$
$\therefore\ larg = \left|uMat[3][2] - vMat[2][3]\right| = 0.000000$

(d) $uMat[3][3] = 272.142883$
$uMat[3][3] = 272.142883$
$vMat[3][3] = 272.142852$
$\left|uMat[3][3] - uMat[3][3]\right| = 0.000000 = larg$
$\therefore\ larg$ remains 0.000000

In step (c) above, because of the seeded error, the program compared $\left|uMat[3][2] - uMat[2][3]\right|$ with *larg*. Since the (erroneous) change in temperature at this point exceeded *larg*, the latter was replaced by $\left|uMat[3][2] - vMat[2][3]\right|$, which was 0. Now that *larg* had become zero, it was less than the stopping criterion, and hence the iteration ended. The iteration for the correct version on the mesh grid G_1 happened to end when the number of iterations was 44, so that results in both the correct and faulty programs agreed. This was also the case for the mesh grid G_2.

In the mesh grid G_3, *larg* became 0 when the faulty version reached the 54th iteration, and hence the program ended. Since G_3 provided a better precision, however, the correct program did not end at the same number of iterations. As a result of more iterations, the computation improved further, and hence the temperature results in the correct program were slightly better than those of the faulty version. In any case, both results were still fairly close to each other because the correct program terminated soon after 54 iterations.

3.2. Testing with Special Test Cases

In this section, we propose several special cases to test the faulty program with a view to detecting the injected error:

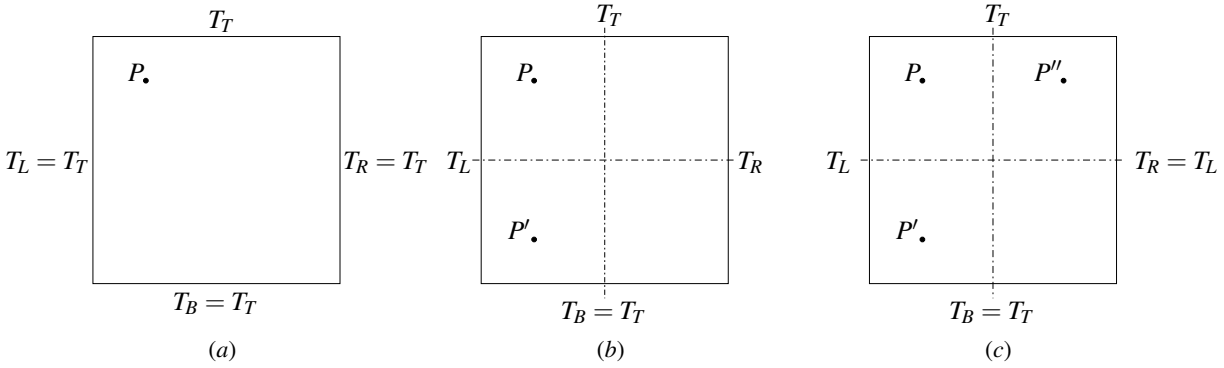


Figure 2. Testing of Special Cases

(*i*) The temperatures at all four edges are identical. In this case, every point on the plate should have the same temperature. Figure 2(*a*) illustrates this situation. Given $T_L = T_R = T_T = T_B$, the temperature at any point P on the plate should be equal to the temperature on the boundary.

(*ii*) Given symmetric boundary conditions in a square plate, we should obtain a symmetric temperature distribution. Let us illustrate this situation by Figure 2(*b*). If $T_T = T_B$, the points P and P' are symmetric with respect to the horizontal axis, so that the temperature at P is identical to that at P'.

(*iii*) Suppose the boundary condition is symmetric not only with respect to the horizontal axis but also with respect to the vertical axis. Let P and P' be points symmetric to the horizontal axis, and P and P'' be points symmetric to the vertical axis. Then, the temperatures at P, P', and P'' should be equal. This is illustrated by Figure 2(*c*).

We have conducted tests covering all these special cases. Unfortunately, they cannot reveal the error in the faulty program.

3.3. Testing with the Metamorphic Method

We would like to discuss how to apply metamorphic testing to the *PDE* program. Consider the mesh grids $(G_i)_{i=1,\,2,\,\ldots,\,5}$ we introduced earlier. Let $T_{G_i}(P)$ be the temperature at point P determined using the mesh grid G_i, and $T(P)$ be the solution of the partial differential equation at P. Let G_i, G_j, and G_k be any three mesh grids. According to the reasoning at the beginning of Section 3, which is based on [14, 19],

$$\begin{aligned} &G_i \subset G_j \subset G_k \;\rightarrow \\ &\quad \big|T_{G_k}(P) - T(P)\big| \le \big|T_{G_j}(P) - T(P)\big| \\ &\quad \le \big|T_{G_i}(P) - T(P)\big|. \end{aligned} \tag{1}$$

Unfortunately, in the absence of a test oracle, $T(P)$ cannot be found easily. To solve the problem, we would like to look for another relation that does not involve $T(P)$. We propose a metamorphic relation

$$\begin{aligned} R_{PDE}:\ G_i \subset G_j \subset G_k \;\rightarrow\; & \\ & T_{G_i}(P) \le \min\{T_{G_j}(P),\, T_{G_k}(P)\} \text{ or} \\ & T_{G_i}(P) \ge \max\{T_{G_j}(P),\, T_{G_k}(P)\}. \end{aligned}$$

This can be proved as follows:

Assume the contrary. Without loss of generality, suppose $T_{G_j}(P) < T_{G_i}(P) < T_{G_k}(P)$.

Case (a): $T(P) \le T_{G_i}(P)$. We have

$$\big|T_{G_k}(P) - T(P)\big| > \big|T_{G_i}(P) - T(P)\big|,$$

thus contracting relation (1).

Case (b): $T(P) > T_{G_i}(P)$. In this situation,

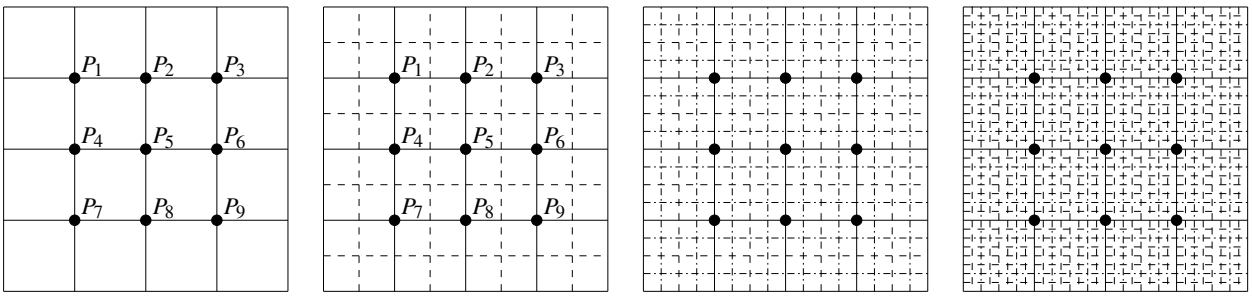
$$\big|T_{G_j}(P) - T(P)\big| > \big|T_{G_i}(P) - T(P)\big|,$$

which also contracts relation (1). ■

R_{PDE} will be the metamorphic relation that we would like to test against the implemented program. If the program does not exhibit this relation, we will have good reasons to suspect a fault.

We check the temperatures in the same 9 points $(P_i)_{i=1,\,2,\,\ldots,\,9}$ using mesh grids G_1, G_2, …, and G_5 such that $G_1 \subset G_2 \subset \ldots \subset G_5$. Figure 3 illustrates the cases for G_1 to G_4.

The complete result is shown in Table 1. An element in the ith row and jth column of the table represents the temperature $T_{G_j}(P_i)$ at point P_i calculated using mesh grid G_j. Consider point P_5. Although $G_3 \subset G_4 \subset G_5$, we have $T_{G_5}(P_5) < T_{G_3}(P_5) < T_{G_4}(P_5)$. This contradicts the metamorphic relation R_{PDE} derived from relation (1). Similarly, at points P_7, P_8, and P_9, we also have $T_{G_5}(P_i) < T_{G_3}(P_i) < T_{G_4}(P_i)$, $i = 7,\ 8,\ 9$, which also contradicts R_{PDE}. Hence, we should suspect a fault in the program.



G_1: 3×3 Mesh Grid G_2: 7×7 Mesh Grid G_3: 15×15 Mesh Grid G_4: 31×31 Mesh Grid

Figure 3. Examples of Mesh Grids for the Square Plate

Point	T_{G_1}	T_{G_2}	T_{G_3}	T_{G_4}	T_{G_5}	*Metamorphic Relation Violated?*
P_1	107.8571	106.7156	106.3443	106.2723	105.6029	No
P_2	105.8929	104.7886	104.3757	104.2979	103.3571	No
P_3	75.7143	74.1261	73.6241	73.5184	72.8407	No
P_4	175.5357	174.0560	173.4716	173.3431	172.3887	No
P_5	190.0000	190.0000	**189.9252**	**189.9529**	**188.6572**	**Yes**
P_6	136.9643	134.4735	133.5401	133.3156	132.3367	No
P_7	304.2857	305.8740	**306.3011**	**306.4345**	**305.8165**	**Yes**
P_8	341.6071	346.6820	**348.4010**	**348.9104**	**348.1195**	**Yes**
P_9	272.1429	273.2844	**273.5808**	**273.6807**	**273.0543**	**Yes**

Table 1. Results of Metamorphic Testing

We note that the failures resulting from the fault are very subtle. They vary from a relative error of 0.2% at point P_7 to 0.7% at point P_5. Even though the faulty statement has been executed repeatedly within a loop for some time, the incorrect computation does not give rise to an obvious anomaly. They may not be detected even in the presence of an approximate test oracle.

4. Conclusion

Because of the lack of analytical solutions, most partial differential equations are usually solved by numerical methods. Owing to the various reasons as explained in Section 1, there may not be a test oracle for such numerical methods. Special test cases may help us reveal some errors, but they cannot detect the subtle errors such as that described in this paper. Instead, we identified a metamorphic relation and conducted metamorphic testing. We managed to identify the fault in the program.

We have demonstrated how the metamorphic testing can help alleviate the oracle problem in the testing of numerical software. Although we have only presented a simple case study to illustrate the usefulness of our metamorphic relation, the relation identified in the paper should also be applicable to other numerical methods that yield better approximations on the refinement of grid points or step sizes. We shall study further applications in our future research.

References

[1] *IMSL Numerical Libraries*. Visual Numerics, Houston, Texas. Available at "`http://www.vni.com/products/imsl`".

[2] *The NAG Fortran Library Manual*. The Numerical Algorithm Group Ltd., Oxford, UK, 2001.

[3] L.M. Adleman, M.-D. Huang, and K. Kompella. Efficient checkers for number-theoretic computations. *Information and Computation*, 121 (1): 93–102, 1995.

[4] P.E. Ammann and J.C. Knight. Data diversity: an approach to software fault tolerance. *IEEE Transactions on Computers*, 37 (4): 418–425, 1988.

[5] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 1990.

[6] M. Blum and S. Kannan. Designing programs that check their work. *Journal of the ACM*, 42 (1): 269–291, 1995.

[7] B. Butler, M. Cox, A. Forbes, S. Hannaby, and P. Harris. A methodology for testing classes of approximation and optimisation software. In

Proceedings of the IFIP WG 2.5 Working Conference on the Quality of Numerical Software: Assessment and Enhancement, pages 138–151. Chapman and Hall, London, 1997.

[8] F.T. Chan, T.Y. Chen, S.C. Cheung, M.F. Lau, and S.M. Yiu. Application of metamorphic testing in numerical analysis. In *Proceedings of the IASTED International Conference on Software Engineering* (*SE '98*), pages 191–197. ACTA Press, Calgary, Canada, 1998.

[9] S.C. Chapra and R.P. Canale. *Numerical Methods for Engineers: with Software and Programming Applications*. McGraw-Hill, New York, 2002.

[10] T.Y. Chen, S.C. Cheung, and S.M. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01. Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, 1998.

[11] T.Y. Chen, T.H. Tse, and Z. Zhou. Fault-based testing in the absence of an oracle. In *Proceedings of the 25th Annual International Computer Software and Applications Conference* (*COMPSAC 2001*), pages 172–178. IEEE Computer Society, Los Alamitos, CA, 2001.

[12] T.Y. Chen, T.H. Tse, and Z. Zhou. Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2002*). *ACM*, New York, NY, 2002.

[13] W.J. Cody and W. Waite. *Software Manual for the Elementary Functions*. Prentice Hall, Englewood Cliffs, New Jersey, 1980.

[14] C.F. Gerald and P.O. Wheatley. *Applied Numerical Analysis*. Addison Wesley, Reading, Massachusetts, 1999.

[15] W.E. Howden. A survey of dynamic analysis methods. In *Software Testing and Validation Techniques*, E.F. Miller and W.E. Howden, editors. IEEE Computer Society, New York, NY, 1981.

[16] M. Mu. Software testing and evaluation in large-scale scientific applications. In *Proceedings of the IFIP WG 2.5 Working Conference on the Quality of Numerical Software: Assessment and Enhancement*, pages 330–332. Chapman and Hall, London, 1997.

[17] J.C.T. Pool. Is numerical software relevant? Is it too late to worry about quality? In *Proceedings of the IFIP WG 2.5 Working Conference on the Quality of Numerical Software: Assessment and Enhancement*, pages 3–11. Chapman and Hall, London, 1997.

[18] D.L. Powers. *Boundary Value Problems*. Academic Press, San Diego, CA, 1999.

[19] J.R. Rice. *Numerical Methods, Software, and Analysis*. Academic Press, Boston, 1993.

[20] C.W. Ueberhuber. *Numerical Computation: Methods, Software, and Analysis*. Springer-Verlag, Berlin, 1997.

[21] E.J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25 (4): 465–470, 1982.