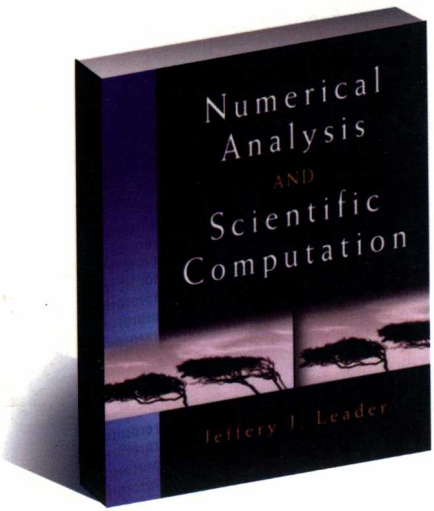
PEARSON
Addison Wesley

大学计算机教育国外著名教材系列（影印版）

Numerical Analysis and Scientific Computation

数值分析与科学计算

Jeffery J. Leader 著



清华大学出版社

大学计算机教育国外著名教材系列（影印版）

Numerical Analysis and Scientific Computation

数值分析与科学计算

Jeffery J. Leader

Rose–Hulman Institute of Technology

清华大学出版社
北京

北京市版权局著作权合同登记号　图字 01-2007-2622

图书在版编目(CIP)数据

数值分析与科学计算：英文/（美）里德（Leader, J.J.）著. —影印本. —北京：清华大学出版社，2008.5
（大学计算机教育国外著名教材系列）
书名原文：Numerical Analysis and Scientific Computation
ISBN 978-7-302-17274-1

Ⅰ. 数…　Ⅱ. 里…　Ⅲ. ①计算机辅助计算：数值计算－英文②计算辅助计算－软件包, MATLAB－英文
Ⅳ. O241　TP391.75

中国版本图书馆 CIP 数据核字（2008）第 042083 号

责任印制： 李红英

出版发行： 清华大学出版社　　**地　　址：** 北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　　编：** 100084
社　总　机： 010-62770175　　**邮　　购：** 010-62786544
投稿与读者服务： 010-62776969, c-service@tup.tsinghua.edu.cn
质　量　反　馈： 010-62772015, zhiliang@tup.tsinghua.edu.cn

印 刷 者： 北京密云胶印厂
印 刷 者： 北京市密云县京文制本装订厂
发 行 者： 全国新华书店
开　　本： 185×230　　**印张：** 38
版　　次： 2008 年 5 月第 1 版　　**印　　次：** 2008 年 5 月第 1 次印刷
印　　数： 1~3000
定　　价： 59.00 元

To my parents,
Dennis and Jeanne Leader

Preface

Numerical analysis is the study of algorithms for the problems of continuous mathematics. Lloyd Trefethen (1992)

Elementary numerical analysis is a marvelously exciting subject. With a little energy and algebra you can derive significant algorithms, code them up, and watch them perform. G. W. Stewart (1998)

THIS BOOK REPRESENTS the numerical analysis course I have taught for the past nine years, first as a required course for sophomore engineering majors and currently as an elective course for juniors and seniors in mathematics, computer science, physics, and engineering. It has been strongly influenced by my summertime computational consulting work at the U.S. Army Research Laboratory (Adelphi, MD), Naval Surface Warfare Center (Dahlgren, VA), and Air Force Research Laboratory (Dayton, OH), and other consulting and research experience; it has also been influenced by conversations with my colleagues and of course by other texts I have turned to over the years.

I hope that what sets my text apart from other introductory numerical analysis texts are the following features:

- Quick introduction to numerical methods, with roundoff error and computer arithmetic deferred until students have gained some experience with real algorithms
- Modern approach to numerical linear algebra
- Explanation of the numerical techniques used by the major computational programs students are likely to use in practice (especially MATLAB, but also Maple and the Netlib library)
- Appropriate mix of numerical analysis theory and practical scientific computation principles
- Greater than usual emphasis on optimization

- Numerical experiments so students can gain experience
- Efficient and unobtrusive introduction to MATLAB

The MATLAB material is introduced in optional subsections at the end of each section. If you'd like your students to learn MATLAB as part of this course, these subsections provide a way to slowly introduce them to it and a chance for them to engage in actual numerical experimentation to build their intuition about scientific computing. If you're not using MATLAB in your course, you may encourage your students to skim these subsections for the extra material and foreshadowing they sometimes contain, but you may use the text without using MATLAB. I have used this approach to teach MATLAB in my classes and have had success with it.

My experience in scientific computing has been that almost every problem reduces, at some level, to a problem in root-finding, nonlinear optimization, or numerical linear algebra, and that the first two cases frequently involve the solution of a linear system. The text starts with root-finding (chapter 1), principally in one dimension, because students will be familiar with it from the calculus. It then moves straight into numerical linear algebra (chapters 2 and 3) because of its crucial importance and so that this material will be available as needed in later chapters. Chapter 4 introduces polynomial interpolation as a tool for deriving other methods and splines as a tool for representing curves given in terms of data. Chapter 5 introduces the basic techniques of numerical quadrature, emphasizing those algorithms employed by MATLAB and Maple. Quadrature of ODEs is given a similar treatment in chapter 6. Nonlinear optimization is covered in chapter 7, and some basic ideas and methods of approximation theory are discussed in chapter 8.

I try to avoid falling into the trap of first introducing a completely new mathematical idea and then teaching computational techniques for it. If students don't already know the mathematical problems—root-finding, finding maxima and minima, differentiation, and integration from the calculus, and first-order ODEs from the calculus or an ODEs course—it will be hard for them to appreciate why they need to know numerical methods for them. Hence I have avoided teaching the Fourier transform, *then* its discrete version the DFT, and then the computational method known as the FFT, for example. However, I have strayed from this principle to introduce the QR and singular value decompositions, as these are not yet standard in a matrix or linear algebra course.

In class, I ask students for ideas about approaching a problem like root finding or about improving a method we have already seen. To mimic that approach in the text, in several sections I briefly discuss alternate methods to the main one I introduce and develop in that section. These brief discussions offer students alternatives I want them to know about, even if we do not discuss them in detail in class or in the text. By the end of the course, these are the kinds of ideas I want them to start suggesting, such as replacing linear approximations with quadratic ones, for example. If you prefer to skip these brief discussions, of course you may.

In emphasizing the ideas behind methods and the mixing of methods I hope to encourage students to feel that they "own" these methods. Too often students feel that they cannot tweak existing algorithms or software, when in fact, adding heuristics is often necessary. In my experience students need to be told that it's OK to fiddle with the methods. That's actually one of the big reasons that people who are only going to *use*, not *write*, numerical software need a course like this; they need to know enough of the

ideas to be able to pick good initial guesses, to approximate needed derivatives, to select the right method or the right parameters.

Over the past decade or so the practice and teaching of numerical analysis have undergone a number of changes. Fewer people write significant amounts of code; they rely more on packages like MATLAB, the similar Octave, GAUSS, and more specialized packages. Also, people are solving bigger and bigger problems, thanks to advances in computer hardware and the availability of powerful software from Netlib, NAG, and so on. This leads to a need for a greater understanding of the ideas behind algorithms and how to tweak them to get them to converge in a timely fashion—choosing proper initial guesses, mixing methods intelligently, adjusting algorithmic parameters—but less need for understanding the algorithms in sufficient detail to write professional level code. (In fact, I believe that writing code should be left to *teams* of experts.) There is greater emphasis on *scientific computation* as opposed to *numerical analysis*.

On the pedagogical side, at many institutions, the theoretical content of calculus courses has been reduced and computer algebra systems have been introduced. This creates a two-fold challenge: First, instructors must work around the students' lesser analytical background, and second, students used to seeing a computer algebra system spit out the answers to all their problems must be convinced of the need for *numerical*, as opposed to *symbolic*, techniques. Students today come to a numerical analysis course with less numerical computing experience than before, thanks to computer algebra systems. This book aims to respond to these pedagogical changes by emphasizing ideas in a way that appeals to geometric thinking (linear and quadratic approximations, for example) and by providing examples of the types of things that cannot be done by computer algebra systems and hence the need for numerical methods.

I mention the possibility of parallelization for several methods but do not develop parallel algorithms in detail. I do not believe a first course in numerical analysis is the place for emphasizing parallel computing, although students should be made aware of its existence. Even if students have access to such a machine, programming it is likely beyond their ability.

This text is *not* intended to be a reference. It is meant to be worked through by students so they learn the subject. I try to avoid dry statements of theorems in favor of rigorous derivations that show the ideas behind the methods. I do not believe that students can appreciate many of the theoretical aspects of the material until they have some experience with numerical computing—all the more because of their experience with computer algebra systems. This text aims to produce students who can competently use standard computational software, including choosing the right method for the job and advising others on the major benefits and pitfalls of various methods. I have had success with this in my current position and hope it works for you in yours.

Each section is meant to be one lecture, although you may find that you need more or less time. The Problems sections contain basic problems that can be used to ensure understanding and do not require new computational resources beyond those required in the preceding sections. The MATLAB subsections introduce MATLAB commands and explore the material through numerical experimentation. Students can use these subsections as self-teaching MATLAB tutorials. I ask students to hand in their output from the `diary` command and to annotate it with brief comments on the results. Some of the Additional Problems may require new computing tools equivalent to those covered in the MATLAB subsections. Additional Problems numbered 10 and above may

be more challenging; the last two or three Additional Problems usually are relatively difficult.

The text contains more material than can be covered in a single quarter or semester. In a ten-week quarter I typically cover the following sections: 1.1–1.9; 2.1–2.9; 3.1–3.6; 4.1–4.4; 5.1–5.3, plus ideas from 5.6 and 5.7; 6.1–6.3, plus ideas from 6.5 and 6.6; and 7.1–7.3 (plus 7.4 and 7.5 if time permits). The course I teach is oriented somewhat toward numerical linear algebra because I believe so many other problems, such as the numerical solution of PDEs, rely on a knowledge of this material, and that many of the problems in those other areas involve getting a linear system solver to work well. More material from chapters 5 and 6 could be included at the expense of material from the latter half of chapter 2. Because of dependencies it will probably be necessary to cover 1.1–1.4, 1.7, 2.1–2.3, and 4.1 in any event.

Prerequisites are a year-long course in the calculus, the basics of matrix algebra, and for chapter 6 the basics of first-order ODEs. Some of the material in chapter 8 is at a more advanced level and requires additional mathematical maturity. Programming experience is not required (if MATLAB is to be used).

My father-in-law wrote in the preface to one of his books that 'The making of a book is the work of many hands.' I have found this to be true. I want to thank Addison-Wesley for publishing this book and to acknowledge in particular Cindy Cody, Joe Vetere, and RoseAnne Johnson at Addison-Wesley; Chris Miller and her group at TechBooks; Louise Gache, the copyeditor; Michael Brown, who wrote the solutions, and the several accuracy checkers; and the many other individuals who have helped make this book.

I also want to acknowledge the helpful comments I have received from a number of anonymous reviewers who suggested changes to the order of presentation, commented on the clarity of various passages, and pointed out errors and omissions. Four years' worth of numerical analysis students at Rose-Hulman Institute of Technology have used the manuscript and were of great help in refining the material, and I gratefully acknowledge their assistance. I wish I could name them all here. Any errors that remain are of course my own.

My colleagues at Rose-Hulman Institute of Technology were very supportive during the time I wrote this book; I want to thank in particular S. Allen Broughton, Ralph P. Grimaldi, and Robert Lopez (now at Maplesoft) for their encouragement and advice.

Over the years I've benefitted from the chance to study and work with a number of numerical analysts who have helped me grow in my understanding of the field. I especially want to acknowledge my advisor, Philip J. Davis, of Brown University; David Gottlieb, also of Brown University; and Bill Gragg, of the Naval Postgraduate School.

I also gratefully thank two people who have been very helpful to me in a great many ways over the course of my career: Robert L. Borrelli and Courtney S. Coleman, both of Harvey Mudd College.

Last but not least I thank my wife, Meg, for her help on this project, and I also thank her and our children Derek and Corrinne for their patience while their father worked long hours on this book.

Jeffery J. Leader
Terre Haute, IN

Contents

1 Nonlinear Equations

1.1 Bisection and Inverse Linear Interpolation

TIME AND TIME AGAIN the solution or simulation of a scientific or engineering problem results in the need to solve either a root-finding problem or an optimization problem. In the former case we seek the solution of an equation or set of equations; in the latter case we seek the point(s) where a function takes on its maximum or minimum value. Even when our goal is to fit a curve to some experimental data or numerically solve a differential equation, for example, we almost always reduce the problem to one of the two types listed above. In this chapter we look at the problem of finding a root of a nonlinear equation; in Chapters 2 and 3 we will look at linear systems of equations, and optimization problems in Chapter 7.

Root-Finding

The root-finding problem is to find a solution x^* of $f(x) = 0$. (There may be many solutions, but we are only seeking any one of them.) The solution is called a **root** of the equation or a **zero** of the function f. For special cases there are often special approaches: The quadratic formula applies if f is a quadratic, and the zeroes of $\sin(x)$ are common knowledge. Sooner or later, however, it becomes necessary to solve a problem that does not fit into a known special case. The simplest examples are equations like

$$\cos(x) - x = 0$$

and polynomials of degree 5 or higher. (There is a cubic formula for degree 3 polynomials and a quartic formula for degree 4 polynomials, but it has been shown that there can be no similar formula for polynomials of degree 5 or higher. Unless a factorization is obvious, numerical methods must be employed.) A more complicated but quite common case is that of finding where the solution of a differential equation passes through zero, when the differential equation itself must be solved numerically.

Using trial and error is one possibility. There is nothing inherently wrong with this approach, but it lacks a theory that predicts how rapidly it will find a solution (to within a given tolerance), and it is difficult to automate. In practice, different root-finding problems may need to be solved tens or hundreds of times within a single run of a program. (For example, this might be true of a computer-aided design package that must determine where various curves intersect.) Because of this we will need to find methods that are *fast, reliable,* and *easily implemented,* ideally without the need for a human to make a judgment at any stage of the process.

Exhaustive Search

The first method we will consider is the method of **exhaustive search** (also called **direct, graphical,** or **incremental search**). Suppose that f is continuous on some (not necessarily finite) interval. By the Intermediate Value Theorem, if we can find two points a, b such that $f(a)$ and $f(b)$ have opposite signs, then a zero of f must lie in (a, b). One way to find such a pair of points is to pick some x_0, an initial guess as to the location of a root, and successively evaluate the function at

$$x_0,\ x_1 = x_0 + h, \quad x_2 = x_0 + 2h, \quad x_3 = x_0 + 3h, \ldots,$$

where $h > 0$ is called the **step size** (or **grid size**). When a change of sign is detected, we say that we have **bracketed** a root in this interval of width h. (The interval $[x_i, x_{i+1}]$ over which the change of sign occurs is called a **bracket;** see Fig. 1.1.) At this point we may repeat the process over the smaller interval $[x_i, x_{i+1}]$ with a smaller h to bracket the root more precisely.

The exhaustive search method is equivalent to plotting the function and looking for an interval in which it crosses the x-axis. This is very inefficient, so let's look for a better approach.

Suppose that we have found, by any means, a bracket $[a, b]$ for a zero of a continuous function. Rather than searching the entire interval with a finer step size, we might reason that the midpoint

$$m = \frac{a+b}{2}$$

is a better estimate of the location of the true zero x^* of f than either a or b. After all,

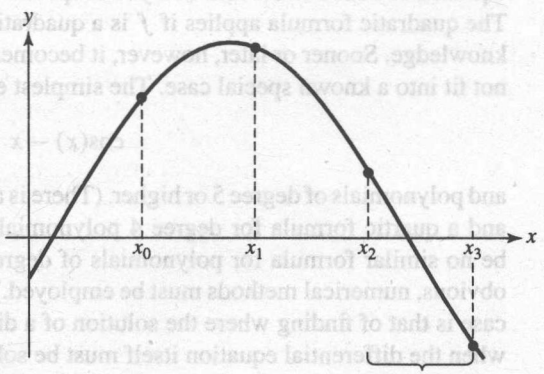


Figure 1.1 Bracketing a Zero.

$|a - x^*|$ could be as large as the width $w = (b - a)$ of the interval if x^* is near b, and similarly for $|b - x^*|$, but

$$|m - x^*| \leq \frac{1}{2}w$$

since x^* lies either in the interval to the right or to the left of m (except in the extremely unlikely case that $x^* = m$). This is the idea behind the **bisection method** (or **binary search**): If f is continuous in the region of interest and $[x_0, x_1]$ is a bracket, that is, $f(x_0)f(x_1) < 0$, then we set

Bisection

$$x_2 = \frac{x_0 + x_1}{2}$$

and compute $f(x_2)$. If $f(x_2) = 0$, we are done; otherwise either $f(x_2)$ and $f(x_0)$ have opposite signs, in which case $[x_0, x_2]$ is a new bracket half the size of the previous one, or $f(x_2)$ and $f(x_1)$ have opposite signs, in which case $[x_2, x_1]$ is a new bracket half the size of the previous one. In either case we have reduced our uncertainty as to the location of the true zero x^* by 50% at the cost of a single new function evaluation (namely the computation of $f(x_2)$). We may now repeat this process on the new interval, finding its midpoint x_3 and then a smaller bracket with x_3 as an endpoint, and so on, until a sufficiently narrow bracket is obtained.

Example 1.1.1 Consider the function $f(x) = \cos(x)$, which has a zero at $\pi/2 \doteq 1.5708$. (The symbol $\doteq$ means that the indicated value is correctly rounded to the number of significant figures given.) Since $f(1) \doteq 0.5403$, $f(2) \doteq -0.4161$, and f is continuous, the interval $[1, 2]$ is a bracket. Its midpoint is $x_2 = 1.5$. Since

$$f(1.5) \doteq 0.0707,$$

we have $f(1.5)f(2) < 0$ and so we replace $x_0 = 1$ with $x_2 = 1.5$, meaning that $[1.5, 2]$ is our improved bracket. The next midpoint is $x_3 = 1.75$, giving

$$f(1.75) \doteq -0.1782$$

so that $[1.5, 1.75]$ is the new bracket. If we had to stop now, our estimate of the location of the true zero could be any value in $[1.5, 1.75]$; choosing its midpoint $x_4 = 1.625$ minimizes the worst-case error. ■

Unless we have the misfortune to choose the wrong interval at some step because a rounding error makes a positive value negative, or vice versa, this method must converge to some zero of f in the initial bracket. That is, as the iteration count k increases, we must have

$$\lim_{k \to \infty} x_k = x^*$$

for some x^* that lies in the initial bracket and for which $f(x^*) = 0$. Furthermore, since the width of the bracket is halved at each step, we can predict how long it will take to achieve any desired precision. If the width of the initial interval is w, then after the first bisection step the width of the new bracket is $w/2$, and after the second step it is $w/4$.

In general, after n steps the width of the resulting interval is equal to $w/2^n$. So to reduce an interval of initial width 1 to an interval of width 10^{-4}, we would need n to be large enough that

$$\begin{aligned} \frac{1}{2^n} &\le 10^{-4} \\ 10^4 &\le 2^n \\ n &\ge \log_2(10^4) \\ &\doteq 13.2877. \end{aligned}$$

So, $n = 14$ iterations would suffice. Since the error is at most 10^{-4}, the answer should have four correct figures after the decimal place (possibly off by one unit in the fourth decimal place). Of course, although this guarantees that the width of the final interval is 2^{-14} (note that $2^{-14} \le 10^{-4} \le 2^{-13}$) and that its midpoint will be an estimate good to within 2^{-15}, the actual error may be much smaller. In fact, performing 14 iterations of the method on $f(x) = \cos(x)$ with the initial bracket $[1, 2]$ and using the midpoint $x_{16} \doteq 1.57077$ of the final interval as our estimate of the true zero $\pi/2 = 1.57079\ldots$ gives an actual error of 2.6×10^{-5} as compared to the bound of 3.1×10^{-5}. The error bound represents a worst-case scenario. Often our results will be considerably better.

The sequence generated by the bisection method is guaranteed to converge to a root; exhaustive search is not (after all, it could step right over a pair of closely spaced roots if h is not sufficiently small; see Fig. 1.2). Bisection will also be much faster in general. However, we have paid a price for these advantages: Bisection requires us to find an initial bracket, whereas exhaustive search requires only a single initial guess lying to the left of the presumed root. Finding that initial bracket may well require an initial graphical search (or trial and error).

Consider again the function $f(x) = \cos(x)$, but now suppose that we have found the initial bracket $[0, 1.6]$ instead, for which we have $f(0) = 1.0000$ and $f(1.6) \doteq -0.0292$. Bisection will work, but noting that $|f(1.6)|$ is very much smaller than $|f(0)|$ might

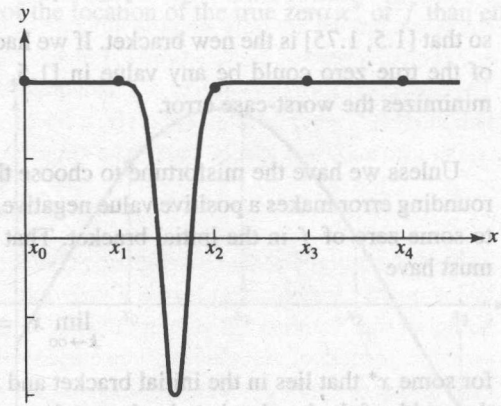


Figure 1.2 Exhaustive Search Misses a Blip.

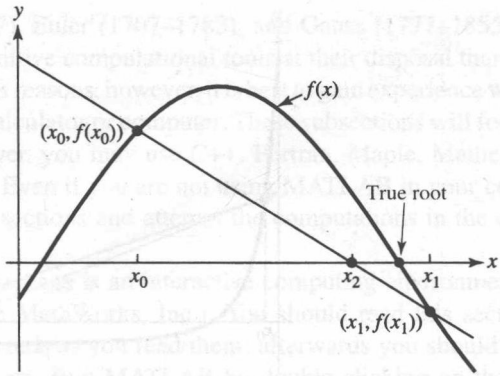


Figure 1.3 Inverse Linear Interpolation.

incline us to choose a new point much nearer $x_1 = 1.6$ than $x_0 = 0$. If we choose, say, $x_2 = 1.4$, then $f(x_2) \doteq 0.1700$ and so [1.4, 1.6] is a new, smaller bracket; the bisection bracket would have been [0.8, 1.6] which is four times as wide. This suggests another method, which is known as **(inverse) linear interpolation** (also called **false position** or ***regula falsi***): Given a continuous function f and a bracket $[x_0, x_1]$ for a zero of f, fit a straight line to the points $(x_0, f(x_0))$ and $(x_1, f(x_1))$ (see Fig. 1.3). This line is said to interpolate f at these points, and if f is approximately linear over the interval, then the zero of the line should be a good estimate of the zero of the function. (The method is called *inverse* interpolation because we are using the interpolated line to find an x value, not a y value as usual; we write x as a linear function of y.) Since the y-values are of opposite sign, $f(x_0)$ is not equal to $f(x_1)$ and so the equation of the line may be written in slope-intercept form as

Inverse Linear Interpolation

$$x = \frac{x_1 - x_0}{f(x_1) - f(x_0)} y + \left(x_1 - \frac{x_1 - x_0}{f(x_1) - f(x_0)} f(x_1) \right) \tag{1.1}$$

and may be solved for the x-intercept x_2 simply by setting $y = 0$:

$$x_2 = x_1 - f(x_1) \frac{x_1 - x_0}{f(x_1) - f(x_0)}. \tag{1.2}$$

As before, either $[x_0, x_2]$ or $[x_2, x_1]$ is a new, smaller bracket. We now iterate until some **convergence criterion** is achieved, that is, until we meet some specified criterion for deciding that we are sufficiently close to the true answer. The width of the interval may not go to zero (if the approach to the root of the equation is one-sided; see Problem 1), so we cannot use that as the only convergence criterion. For the same reason, unless the width of the bracket is very small, the last computed endpoint is generally the best estimate of the location of the root, not the midpoint as before.

Inverse linear interpolation will converge to some zero of the function in the bracket; however, the rate at which it converges will depend on how nearly linear $f(x)$ is near its zero. We know from the calculus that if $f(x)$ is sufficiently differentiable then it is well approximated by a straight line over small intervals. For the type of functions usually

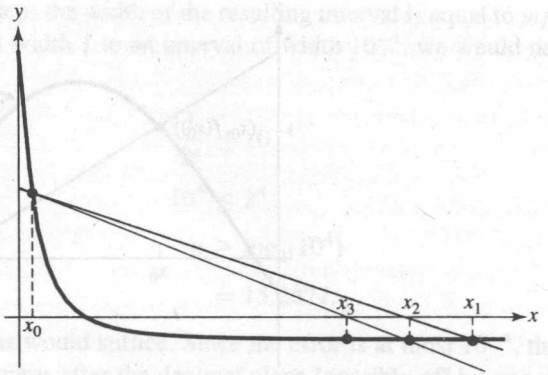


Figure 1.4 Tough Function for Inverse Linear Interpolation.

encountered in practice this means that inverse linear interpolation will usually be faster than bisection. (A function and bracket as in Fig. 1.4 will result in excruciatingly slow convergence until the bracket is very small.) We have given up the guaranteed, slow-but-steady-wins-the-race speed of bisection for a likely but neither guaranteed nor easily predictable improvement.

In the next section we consider a much faster algorithm known as Newton's method. Once again we pay a price for using the method: We require that the function be differentiable as well. Faster methods require more assumptions and offer fewer guarantees.

PROBLEMS 1.1

1. Use four iterations of bisection with the initial bracket $[0.5, 1]$ to approximate the sole positive root of $\cos(x) - x = 0$. Repeat with inverse linear interpolation, using the same initial bracket; note the one-sided approach to the solution.
2. Show that there is a unique real solution of $5x^7 = 1 - 2x$. Use bisection to approximate it to four decimal places. Repeat with inverse linear interpolation.
3. The function $f(x) = \cos(5x)$ has seven zeroes in the interval $[0, 4.5]$. Use bisection and then inverse linear interpolation with that initial bracket (verify that it is in fact a bracket) to find a zero to at least three decimal places. Find the actual error in your estimates using the known locations of the zeroes of the cosine function. Comment on your results.
4. Use bisection on $f(x) = x^2 - 5$ to approximate $\sqrt{5}$ to at least four decimal places. Compare your error estimate to the actual error.
5. Use inverse linear interpolation with the initial bracket $[0.25, 2]$ to approximate a zero of $f(x) = 1/x^3 - 10$ to three decimal places. Repeat with bisection. Comment.

MATLAB 1.1

Numerical analysis is the study and design of methods that may be used to solve the mathematical problems of engineering and the sciences by iterative algorithms in a reasonable time and with desirable error properties.[1] This can be done without access to a computer, as is clear from the names attached to some of these methods—such as

[1] Lloyd Trefethen writes: "Numerical analysis is the study of algorithms for the problems of continuous mathematics."