More Accurate Linear Interpolation For Table Lookup Algorithms

John Forkosh
285 Stegman Parkway #309
Jersey City, NJ 07305
(201) 860−9677

Resume

John Forkosh has an ms in physics, and for the past ten years has been an independent consultant working primarily under VAX/VMS in "C". He developed the algorithm discussed in this article while at the Goddard Institute for Space Studies in New York City, working on the radiative transfer code for a general circulation model of the earth's atmosphere (used to study climatic change, particularly due to increased carbon dioxide resulting from burning fossil fuels).

I. Introduction

Scientific applications often require the very frequent evaluation of complicated functions. To reduce computational overhead, such a function can be defined by a table of its values at various fixed points. Some form of interpolation is then required to estimate values of the function at intermediate points. Linear interpolation is the quickest procedure; however, it is usually the least accurate.

A lookup table for a function f(x) is a sequence of numbers y_i=f(x_i), i=1...n. Using linear interpolation, f(x) is approximated by straight line segments connecting the y_i's. Consider the very simple function $f(x)=x^2$. Because this function is concave (its second derviative f"(x)=2 is always positive) the straight line segments will always lie above the curve, i.e., linearly interpolated values will always be too large.

To avoid such systematic errors, we suggest line segments connecting a sequence of "pseudo−values" y_i* that are determined by minimizing the sum of the integrated square errors attributable to each segment. A least−squares technique is used to determine the appropriate y_i*'s. The result is that the table does not contain exact values for the function f(x) at the points x_i. However, the mean square error due to sampling random points in the function's domain is minimized. For instance, for $f(x)=x^2$ between $-10\leq x\leq 10$ with table values at each integer, the mean square error is reduced by a factor of 6 (the rms error by a factor of 2.45).

Our discussion is pitched at about the level of Numerical Recipes in C, Press et.al., Cambridge University Press 1988. It illustrates not only interpolation (Chapter 3), but also least squares (Chapter 14) and the solution of linear algebraic equations (Chapter 2). The latter arises because, for a table with n values, our analysis expresses the y_i*'s as n simultaneous equations in terms of the y_i's. Complete code is provided so that the reader can apply the algorithm to any function f(x) whatsoever.

II. Functions of a Single Variable

Interpolation techniques represent a function y=f(x) by a table of values $y_i=f(x_i)$, i=1...n. Between y_i and y_{i+1}, standard linear interpolation approximates f(x) by a line segment, which we'll denote by $u_i(x)=a_ix+b_i$, $x_i\leq x\leq x_{i+1}$, with a_i and b_i chosen so that $u_i(x_i)=y_i$ and $u_i(x_{i+1})=y_{i+1}$. This provides exact values of f(x) at the x_i's, but does not constrain the mean square errors

$$\epsilon_i = \frac{1}{x_{i+1}-x_i}\int_{x_i}^{x_{i+1}} (u_i(x)-f(x))^2 dx. \tag{1}$$

In addition, it gives rise to systematic errors when the sign of f"(x) doesn't change between points. If f">0 (concave function) then u−f>0 for all x in the interval (and similarly for a convex function). This situation is illustrated in figure 1a.

Figure 1b illustrates the procedure to be developed. In this case the line segments $u_i(x)$ connect points $u_i(x_i)=y_i*$ and $u_i(x_{i+1})=y_{i+1}*$. The y_i*'s are determined by a least squares technique that minimizes the sum of the error terms given in Eq.(1), so that

$$\epsilon = \sum_{i=1}^{n-1} \epsilon_i; \quad \frac{\partial \epsilon}{\partial y_i*} = 0, \, i = 1...n. \tag{2}$$

This gives n equations for the n unknown y_i*'s.

The work between here and Eq.(8) contains the algebra necessary to formulate these n equations so that we can program their solution. The reader who doesn't care to follow this derivation can skip to Eq.(8) which contains the required result. If you do skip there, note that $y_{i\pm\frac{1}{2}}=f(x\pm\frac{1}{2}\Delta x)$ denotes the middle of each Δx interval.

To keep the analysis tractable we assume that $x_{i+1}-x_i=\Delta x$ is constant. Then the line segments can be written

$$u_i(x) = \frac{1}{\Delta x}\left[(x_{i+1}-x)y_i * + (x-x_i)y_{i+1}*\right], \; x_i \le x \le x_{i+1},$$

and the sum of the error terms given by Eqs.(1) and (2) is

$$\epsilon = \sum_{i=1}^{n-1} \frac{1}{\Delta x} \int_{x_i}^{x_{i+1}} \left\{ \frac{1}{\Delta x}\left[(x_{i+1}-x)y_i * + (x-x_i)y_{i+1}*\right] - f(x) \right\}^2 dx \tag{3}$$

At the interior points $1<i<n$ two terms from the sum in Eq.(3) contain subscript i, so that

$$\frac{\partial \epsilon}{\partial y_i*} = \frac{\partial \epsilon_{i-1}}{\partial y_i*} + \frac{\partial \epsilon_i}{\partial y_i*}, \; 1 < i < n, \tag{4}$$

while at the boundaries i=1 and i=n only one term contributes, so that

$$\frac{\partial \epsilon}{\partial y_1*} = \frac{\partial \epsilon_1}{\partial y_1*}, \; \frac{\partial \epsilon}{\partial y_n*} = \frac{\partial \epsilon_{n-1}}{\partial y_n*}. \tag{5}$$

These derivatives can be evaluated directly as follows:

$$\frac{\partial \epsilon_{i-1}}{\partial y_i*} = \frac{2}{\Delta x^2}\int_{x_{i-1}}^{x_i}\left\{\frac{1}{\Delta x}\left[(x-x_{i-1})y_i*+(x_i-x)y_{i-1}*\right]-f(x)\right\}(x-x_{i-1})dx,$$

$$\frac{\partial \epsilon_{i}}{\partial y_i*} = \frac{2}{\Delta x^2}\int_{x_{i}}^{x_{i+1}}\left\{\frac{1}{\Delta x}\left[(x_{i+1}-x)y_i*+(x-x_i)y_{i+1}*\right]-f(x)\right\}(x_{i+1}-x)dx.$$

Substitute $x_{i-1}=x_i-\Delta x$ and $x_{i+1}=x_i+\Delta x$ to obtain:

$$\frac{\partial \epsilon_{i-1}}{\partial y_i*} = \frac{2}{\Delta x^3}\int_{x_i-\Delta x}^{x_i}\left\{(y_i*-y_{i-1}*)(x-x_i+\Delta x)+y_{i-1}*\Delta x-f(x)\Delta x\right\}(x-x_i+\Delta x)dx,$$

$$\frac{\partial \epsilon_{i}}{\partial y_i*} = \frac{2}{\Delta x^3}\int_{x_i}^{x_i+\Delta x}\left\{(y_{i+1}*-y_i*)(x-x_i-\Delta x)+y_{i+1}*\Delta x-f(x)\Delta x\right\}(x-x_i-\Delta x)dx.$$

Now separate the f(x) term from each integral and substitute $x^*=x-x_i+\Delta x$ on the top line, $x^*=x-x_i-\Delta x$ on the bottom line:

$$\frac{\partial \epsilon_{i-1}}{\partial y_i*} = \frac{2}{\Delta x^3}\int_{0}^{+\Delta x}\left\{(y_i*-y_{i-1}*)x*+y_{i-1}*\Delta x\right\}x*dx* - \frac{2}{\Delta x^2}\int_{x_i-\Delta x}^{x_i} f(x)(x-x_i+\Delta x)dx,$$

$$\frac{\partial \epsilon_{i}}{\partial y_i*} = \frac{2}{\Delta x^3}\int_{0}^{-\Delta x}\left\{(y_{i+1}*-y_i*)x*+y_{i+1}*\Delta x\right\}x*dx* - \frac{2}{\Delta x^2}\int_{x_i}^{x_i+\Delta x} f(x)(x_i-x+\Delta x)dx.$$

The left−hand integrals can be evaluated directly:

$$\frac{\partial \epsilon_{i-1}}{\partial y_i*} = \frac{2}{3}y_i*+\frac{1}{3}y_{i-1}* - \frac{2}{\Delta x^2}\int_{x_i-\Delta x}^{x_i} f(x)(x-x_i+\Delta x)dx, \qquad (6)$$

$$\frac{\partial \epsilon_{i}}{\partial y_i*} = \frac{2}{3}y_i*+\frac{1}{3}y_{i+1}* - \frac{2}{\Delta x^2}\int_{x_i}^{x_i+\Delta x} f(x)(x_i-x+\Delta x)dx. \qquad (7)$$

So far the analysis is exact. To finish the problem we approximate the right−hand integrals using Simpson's rule applied at the midpoint of each interval. Let $y_{i\pm\frac{1}{2}}=f(x\pm\frac{1}{2}\Delta x)$ so that

$$\int_{x_i-\Delta x}^{x_i} f(x)dx \approx \frac{\Delta x}{6}(y_{i-1} + 4y_{i-\frac{1}{2}} + y_i),$$

from which we can evaluate

$$\frac{2}{\Delta x^2}\int_{x_i-\Delta x}^{x_i} f(x)(x - x_i + \Delta x)dx \approx \frac{1}{3}(y_i + 2y_{i-\frac{1}{2}}), \tag{8}$$

$$\frac{2}{\Delta x^2}\int_{x_i}^{x_i+\Delta x} f(x)(x_i - x + \Delta x)dx \approx \frac{1}{3}(y_i + 2y_{i+\frac{1}{2}}). \tag{9}$$

Any desired degree of accuracy can be obtained by using better approximations in Eqs.(6). The results are then substituted back into Eqs.(5), and in this case

$$\frac{\partial\epsilon_{i-1}}{\partial y_i *} \approx \frac{2}{3}y_i * + \frac{1}{3}y_{i-1} * \; - \frac{1}{3}(y_i + 2y_{i-\frac{1}{2}}), \tag{10}$$

$$\frac{\partial\epsilon_{i}}{\partial y_i *} \approx \frac{2}{3}y_i * + \frac{1}{3}y_{i+1} * \; - \frac{1}{3}(y_i + 2y_{i+\frac{1}{2}}). \tag{11}$$

Eqs.(2), (4) and (7) now let us write down n equations for the n unknown y_i*'s in terms of the known y_i's:

$$\begin{aligned} 2y_1 * + y_2 * &= y_1 + 2y_{1+\frac{1}{2}}, & for\, i = 1, \\ y_{i-1} * + 4y_i * + y_{i+1} * &= 2y_{i-\frac{1}{2}} + 2y_i + 2y_{i+\frac{1}{2}}, & 1 < i < n, \\ 2y_n * + y_{n-1} * &= y_n + 2y_{n-\frac{1}{2}}, & i = n. \end{aligned}$$

These results are most easily written in matrix form:

$$\begin{pmatrix} 2 & 1 & . & . & . & . & . \\ 1 & 4 & 1 & . & . & 0 & . \\ . & 1 & 4 & 1 & . & . & . \\ . & . & 1 & 4 & 1 & . & . \\ . & . & . & 1 & 4 & 1 & . \\ . & 0 & . & . & 1 & 4 & 1 \\ . & . & . & . & . & 1 & 2 \end{pmatrix} \begin{pmatrix} y_1* \\ . \\ . \\ y_i* \\ . \\ . \\ y_n* \end{pmatrix} = \begin{pmatrix} y_1 + 2y_{1+\frac{1}{2}} \\ . \\ . \\ 2y_{i-\frac{1}{2}} + 2y_i + 2y_{i+\frac{1}{2}} \\ . \\ . \\ y_n + 2y_{n-\frac{1}{2}} \end{pmatrix} \tag{12}$$

Eq.(8) specifies how to construct a table of y*'s, for linearly interpolating any function f(x), that minimizes (modulo Simpson's rule) the mean square error due to randomly sampling points in the function's domain. However, f(x) must be a function of a single variable. The application for which this alogorithm was originally designed requires interpolation of a function z=f(x,y) of two variables. This more general situation is conceptually quite similar to the one−dimensional case we discussed; however, the required algebra becomes noticeably more complicated. I therefore decided to stop the formal discussion of the algorithm at this point. However, if some unanticipated level of interest arises, I'll be happy to supply CUJ with a short discussion and the necessary code.

III. Discussion of Code

The implementation of Eq.(8) is demonstrated in Listings 1 and 2. Listing 1 contains a "C" function lint1d(x0,dx,n,f,y) that returns a table of y*'s, suitable for one−dimensional linear interpolation, as prescribed by Eq.(8). It simply malloc's a temporary nxn array in which the coefficient matrix is set up, then calls the user−supplied function f() to initialize array y[] with the vector of constants, and finally solves the simultaneous equations, returning the y* solutions to the caller in the same array y[]. Listing 1 also contains a test driver for lint1d(), and some sample output for $f(x)=x^2$ as discussed in the introduction. Listing 2 contains the simultaneous equation solver used by lint1d(). It's a Fortran−to−C port of an old IBM Scientific Subroutine Package routine, simq(), as noted in the code. As you can see, Eq.(8) specifies a very straightforward procedure, and the resulting code for lint1d() is correspondingly simple. The code for simq() is a bit dense, but can be treated as a black box for the purposes of this application.

By supplying the address of your own function to lint1d()'s f−argument, an interpolation table can very easily be constructed to suit any purpose whatsoever. Thus, almost any application that employs one−dimensional linear interpolation at fixed intervals should be able to realize noticeable improvements in accuracy. All it takes is a simple one−line call to lint1d(). Afterwards, interpolation on the improved table is performed totally transparently, so absolutely no application−level recoding is required.

Both listings contain embedded drivers (that are compiled when the #defined flag TESTDRIVE is true) to facilitate module−level testing and to illustrate usage. I've also tried to carefully document the code (except for some of the more elaborate indexing arithmetic required near the bottom of simq()). My personal experience with scientific code is that it frequently tends to be under−documented and also under−robust. The latter problem can be particularly pernicious since the code usually won't blow up, but will simply supply the wrong result due to an inappropriate numerical method or due to some more trivial problem. (For instance, I once worked on a model where the value of π had been inadvertantly entered incorrectly.) Since numerical results are usually not analytically verifiable, and since feedback mechanisms frequently prevent small errors from growing uncontrollably, it may take quite some time (if ever) before a researcher realizes that his results are little more than massaged noise. The only effective procedure to guard against such difficulties is a design that carefully decomposes a complicated calculation into easily describable functional components that can be individually tested for known correct results.