LAWRENCE LIVERMORE NATIONAL LABORATORY

LLNL-TR-406719-REV-5

LIP: The Livermore Interpolation Package, Version 1.6

F. N. Fritsch

January 4, 2016

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

LIP:
The Livermore Interpolation Package

Version 1.6

Frederick N. Fritsch

Physical and Life Sciences Directorate
Physics Division
Condensed Matter Section
EOS and Materials Theory Group

December 2015

(This page deliberately left blank.)

Contents

Introduction

This report describes LIP, the Livermore Interpolation Package. LIP was totally rewritten from the package described in [1]. In particular, the independent variables are now referred to as x and y, since it is a general-purpose package that need not be restricted to equation of state data, which uses variables ρ (density) and T (temperature).

LIP is primarily concerned with the interpolation of two-dimensional data on a rectangular mesh. The interpolation methods provided include piecewise bilinear, reduced (12-term) bicubic, and bicubic Hermite (biherm). There is a monotonicity-preserving variant of the latter, known as bimond. For historical reasons, there is also a biquadratic interpolator, but this option is not recommended for general use. A birational method was added at version 1.3. In addition to direct interpolation of two-dimensional data, LIP includes a facility for inverse interpolation (at present, only in the second independent variable). For completeness, however, the package also supports a compatible one-dimensional interpolation capability. Parametric interpolation of points on a two-dimensional curve can be accomplished by treating the components as a pair of one-dimensional functions with a common independent variable.

LIP has an object-oriented design, but it is implemented in ANSI Standard C for efficiency and compatibility with existing applications. First, a “LIP interpolation object’ is created and initialized with the data to be interpolated. Then the interpolation coefficients for the selected method are computed and added to the object. Since version 1.1, LIP has options to instead estimate derivative values or merely store data in the object. (These are referred to as “partial setup” options.) It is then possible to pass the object to functions that interpolate or invert the interpolant at an arbitrary number of points.

The first section of this report describes the overall design of the package, including both forward and inverse interpolation. Sections 2–6 describe each interpolation method in detail. The software that implements this design is summarized function-by-function in Section 7. For a complete example of package usage, refer to Section 8. The report concludes with a few brief notes on possible software enhancements. For guidance on adding other functional forms to LIP, refer to Appendix B.

The reader who is primarily interested in using LIP to solve a problem should skim Section 1, then skip to Sections 7.1–4. Finally, jump ahead to Section 8 and study the examples. The remaining sections can be referred to in case more details are desired.

Changes since version 1.1 of this document include the new Section 3.2.1 that discusses derivative estimation and new Section 6 that discusses the birational interpolation method. Section numbers following the latter have been modified accordingly. (Note that version 1.2 was not formally documented.) A 1-D example (see Section 8.2) was added at version 1.5. Section 7.9 and Appendix G were added at version 1.6.

1. Overall Package Design

This section describes the overall design of LIP. The organization of the software that implements this design is described in Section 7, below.

1.1. Data

Throughout this report we assume that data has been given for some function f(x,y) on a rectangular mesh $\mathbf{x} = (x_0, x_1, ..., x_{nx-1})$, $\mathbf{y} = (y_0, y_1, ..., y_{ny-1})$. Subscripting is from zero to be consistent with the C code. The data values are $f_{ij} = f(x_i,y_j)$. There are nx×ny data values, (nx–1)×(ny–1) mesh rectangles (boxes). The mesh is arbitrary, except that we require:

$$s_x.x_{i-1} < s_x.x_i, \quad i = 1, \ldots, nx-1; \tag{1.1x}$$

$$s_y.y_{j-1} < s_y.y_j, \quad j = 1, \ldots, ny-1. \tag{1.1y}$$

where

$$s_x = +1 \text{ if } x_0 < x_{nx-1},\ s_x = -1 \text{ if } x_0 > x_{nx-1};$$

$$s_y = +1 \text{ if } y_0 < y_{ny-1},\ s_y = -1 \text{ if } y_0 > y_{ny-1}.$$

In the C code, the data array is one-dimensional, with `data[j*nx+i]` = $f(x_i,y_j)$.

In Version 1.5 the restriction that the independent variable (IV) arrays must be increasing (s_x=+1, s_y=+1) was removed. It is now sufficient that they be strictly monotonic (either increasing or decreasing). This does not apply to the deprecated methods `LIP_QUAD` and `LIP_CUBM`. (Refer to Appendix A for definitions of these interpolation type identifiers.)

NOTE: The following notation is used in subsequent sections:

$$xlow = \min(x_i, x_{i+1});\ \ xhigh = \max(x_i, x_{i+1});\ \ x_{imin} = xlow;\ \ x_{imax} = xhigh.$$

$$ylow = \min(y_j, y_{j+1});\ \ yhigh = \max(y_j, y_{j+1});\ \ y_{jmin} = ylow;\ \ y_{jmax} = yhigh.$$

The coefficients are based at the point with lowest *value*, rather than lowest *index*. (The last two equations in each row define indices imin,imax and jmin,jmax.)

1.2. Compatible univariate interpolation

The original motivation for including univariate interpolation in LIP was to handle the equation of state (EOS) for univariate “cold curves”. For a compatible interpolation method, it is expected that

$$Ft = Fc + Fe + Fi\ , \tag{1.2}$$

where the second letters stand for total (t), cold (c), electronic (e) and ionic (i). Fc is univariate (a function of density only), and the others are bivariate. (Here F = E or P for standard EOS data.) The univariate interpolant needs to be compatible with its bivariate interpolant of the same type (bilinear, bicubic, or biherm), in the sense that if the data for these four functions satisfy (1.2), then so will the interpolants (to as close to machine precision as possible).

The current package contains compatible linear, cubic, and cubic Hermite interpolators. Tests show good consistency for these methods, whereas the bimond interpolant (see Section 4.3) is much less compatible, due to the inherently two-dimensional nature of this method. A univariate rational interpolator was added at version 1.3, but there was no attempt to make it compatible with the birational method. Since the EOP data is available only for Ft, there is no univariate quadratic interpolator.

In the case of univariate data supported by LIP, one of the independent variables is omitted, as well as the associated index on the data array. When ny=0, `data[i]` = $f(x_i)$; when nx=0, `data[j]` = $f(y_j)$.

1.3. Interpolation coefficients

After an interpolation object has been initialized, the normal first step is to select an interpolation method and then compute interpolation coefficients for that method. The resulting object is then ready to be used for the desired application. The full coefficient array contains ncoef×nboxes = ncoef×(nx–1)×(ny–1) numbers, where ncoef increases as the smoothness of the interpolation method increases. See the following table.

Method	**ncoef**	**Function**	**Derivatives**	**Monotonic?**
Bilinear	4	Continuous	Jump discontinuity across mesh lines	Yes
Biquadratic	9	Continuous only at data points	Usually not continuous	No
Bicubic	12	Continuous (due to current derivative approximation; see Section 3.2.1)	Continuous along mesh lines; may be discontinuous across mesh lines	No
Bicubic Hermite	16	Continuous	Continuous across mesh lines	May not be
Bimond	16	Continuous	Continuous across mesh lines	Yes

The univariate piecewise linear interpolant requires two coefficients per mesh cell, whereas the univariate cubic interpolants all use ncoef=4.

1.4. Partial Setup Options

The primary new capability of LIP version 1.1 was the provision for partial setup (not requiring the full coefficient array). Full support for partial setup was available in most cases at version 1.2. This has two motivations. First, and foremost, was the desire to reduce the memory requirements, possibly at the expense of reduced evaluation speed. A secondary motivation was the ability to incorporate other interpolation methods that do not fit naturally into the standard interpolation coefficient mold. Note that the birational method (see Section 6, below) added at version 1.3 is such a method.

Three basic modes are available in version 1.1 and subsequent versions:

(1) *Full coefficient setup*. This was the standard mode in version 1.0.

(2) *Derivatives-only setup*. Approximate derivative values; compute coefficients as needed at evaluation time. This comes in two flavors: (2a) first partial derivatives only, or (2b) first derivatives and twists. The latter is relevant only for bicubic Hermite interpolation (see Section 4, below).

(3) *Data-only setup*. Store data and interpolate directly from the data array.

The storage requirements for each mode are as follows, with ncoef as in previous table:

Mode (1) bivariate: ndata + ncoef × (nx–1) × (ny–1)

Mode (1) univariate: ndata + ncoef × max[(nx–1),(ny–1)]

Mode (2a) bivariate: ndata + 2×nx×ny

Mode (2a) univariate: ndata + max(nx,ny)

Mode (2b) bivariate: ndata + 3×nx×ny

Mode (3): ndata

In the above formulas,

ndata = nx×ny + nx + ny, for bivariate data;
ndata = 2×max(nx,ny), for univariate data.

For example, for bicubic Hermite we see that ratio of mode (1) to mode (2a) storage requirements is approximately 17/3=5.66...

Not all modes are available for all methods. The following table summarizes the availability in LIP version 1.6.

Method	**dimension**	**Mode (1)**	**Mode(2a)**	**Mode(2b)**	**Mode (3)**
Bilinear	bivariate	Yes	N/A[1]	N/A[1]	Yes
	univariate	Yes	N/A[1]	N/A[1]	Yes
Biquadratic	bivariate	Yes	No	No	No

Method	dimension	Mode (1)	Mode(2a)	Mode(2b)	Mode (3)
Bicubic	bivariate	Yes	Yes[2]	N/A	Yes[2]
	univariate	Yes	Yes	N/A	Yes
Bicubic Hermite	bivariate	Yes	Yes	Yes	Yes
	univariate	Yes	Yes	N/A	Yes
Bimond	bivariate	Yes	Yes	Yes	Yes[3]
	univariate	Yes	Yes	N/A	No
Birational	---	N/A	N/A	N/A	Yes

Notes:

1. If a Mode (2) setup is attempted for linear interpolation, Mode (3) results.
2. The two-phase modification (see Section 3.3) is not done in Mode (2) or (3).
3. It is not guaranteed that the function and/or derivative values computed by bimond (see Section 4.3) will be identical to those using other modes.

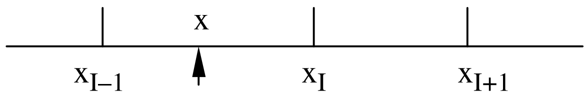
Partial setup (Mode >1) is currently supported for inverse evaluation in all bivariate cases marked “Yes” in the above table. Support for bimond was added at Version 1.4.

1.5. Forward interpolation (evaluation)

Given an array of points (x_{int}, y_{int}) at which interpolation is desired, the first step is to call a table lookup function twice, the first to determine the indices of the x_{int} in **x**, and the second to locate the y_{int} in **y**. This step is independent of the interpolation method.

With any functional form, a nontrivial part of the evaluation for a given (x, y) is determining box indices i and j such that $x_i \le x < x_{i+1}$ and $y_j \le y < y_{j+1}$. The two variables are searched independently. LIP uses a binary search with guess. The index found at one point is saved and used as a guess at the interval index for the next point. (The index for the first point is initialized to zero.) If the defining conditions are already satisfied by the saved index, we are done. If not, the next interval in the direction of the input value is examined. If that test also fails, then a binary search is performed on the remaining part of the data.

For example, let I be the saved x-index. If $x_{I-1} \le x < x_I$, the test $x_I \le x$ will fail. From the direction of the failure, the code will then test for $x_{I-1} \le x$. This succeeds in this case, and the new x-index is i=I–1. If $x < x_{I-1}$, however, a binary search will then be performed in $[x_0, x_{I-1}]$.



In order to handle either increasing or decreasing x-values, this procedure is modified by replacing each test (x_1 @ x_2) by ($s.x_1$ @ $s.x_2$), where s = +1 if $x_0 < x_{nx-1}$ or –1, otherwise. (Here, @ is one of the relational operators ≤, <, >, ≥.) To avoid confusion, most of the rest of this section is written as though $x_i < x_{i+1}$, but similar modifications will handle the decreasing case.

In order to provide thread-safety, the function `lip_lookup` that implements the lookup procedure for a single variable saves no state between calls. It does proceed as indicated, but starts anew for each array it is asked to look up. It returns an array of indices, which are then passed on to the appropriate evaluation routine for a most efficient computation.

The mesh index arrays are partitioned into subsets such that all of the consecutive (x_{int}, y_{int}) lie in the same mesh box (i_{box}, j_{box}), and the interpolant is evaluated at all of those points using a common set of ncoef interpolation coefficients via a call to `lip_evalu_box`. This, in turn, calls a specific evaluator, which depends on the method. The variables are transformed as indicated below and the interpolation coefficients are obtained. If the full coefficient array is available, a pointer is set to the appropriate location in the coefficient array. If partial setup has been implemented, the coefficients for this mesh box are computed. Then the appropriate equation is evaluated. If derivatives are requested, the appropriate derivative formulas are also evaluated, and the values are returned to the calling program. These evaluators are described in more detail below.

A similar procedure is used in the univariate case, but there is, of course, only a single call to `lip_lookup`. The analog of `lip_evalu_box` is called `lip_evalu_cell`.

1.6. Variable transformation

For the standard bivariate forms, the following variable transformations are used to simplify formulas and enhance numerical stability:

$$x = \mathrm{xscale}(x_{int}) = (x_{int} - x_{imin}) / \Delta x_i, \;\; \Delta x_i = x_{imax} - x_{imin}; \qquad (1.3x)$$

$$y = \mathrm{yscale}(y_{int}) = (y_{int} - y_{jmin}) / \Delta y_j, \;\; \Delta y_j = y_{jmax} - y_{jmin}. \qquad (1.3y)$$

Here (x_{int},y_{int}) is the point in the original variables at which interpolation is desired, and (x,y) is the transformed (scaled) point. Note that these scale functions have the property

$$\mathrm{xscale}(x_{imin})=0, \;\; \mathrm{xscale}(x_{imax})=1, \;\; \mathrm{yscale}(y_{jmin})=0, \;\; \mathrm{yscale}(y_{jmax})=1, \qquad (1.4)$$

so that the rectangle $R_{ij}=[x_{imin},x_{imax}]\times[y_{jmin},y_{jmax}]$ is mapped onto the unit square $U=[0,1]\times[0,1]$.

These scaling transformations are performed in the method-dependent functions called by `lip_evalu_box`.

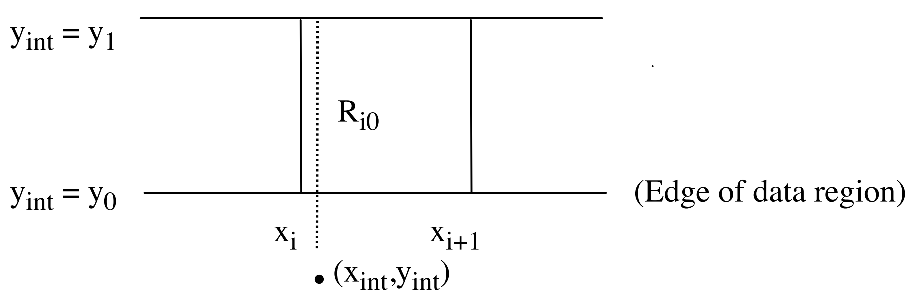
1.7. Extrapolation

If an interpolation point (x_{int},y_{int}) is outside the data mesh (i.e., $x_{int} < x_0$ or $x_{int} > x_{nx-1}$, $y_{int} < y_0$ or $y_{int} > y_{ny-1}$), we are in an "extrapolation" situation. The current LIP evaluators allow two extrapolation options, controlled by argument `extr_flag`.

If `extr_flag=0` and (x_{int},y_{int}) is outside the table, then the value at the nearest boundary point is returned. (That is, no extrapolation is performed.) If derivatives are requested, zero is returned to match the constant behavior of the extrapolant.

On the other hand, if `extr_flag=1` and (x_{int},y_{int}) is outside the table, the value of the function and derivative at the nearest edge point are computed. The linear function determined by these two values is evaluated at (x_{int},y_{int}) for the returned value. (This will be linear in the out-of-range variable and the order requested in the other.) The edge derivatives are returned if requested. If both values are out of range, then the bilinear extrapolant determined by the function and derivative values at the nearest corner is used.

As an example, suppose $x_i < x_{int} < x_{i+1}$ but $y_{int} < y_0$ (extrapolation in y):



In this case, we will have scaled variable values 0<x<1, but y<0. If `extr_flag=0`, then $f(x_{int},y_0)$, the value at (x,0), will be returned. If `extr_flag=1`, then $\partial f(x_{int},y_0)/\partial y$ will also be computed (even if not requested) and $f(x_{int},y_0) + (y_{int}-y_0)\ \partial f(x_{int},y_0)/\partial y$ will be returned.

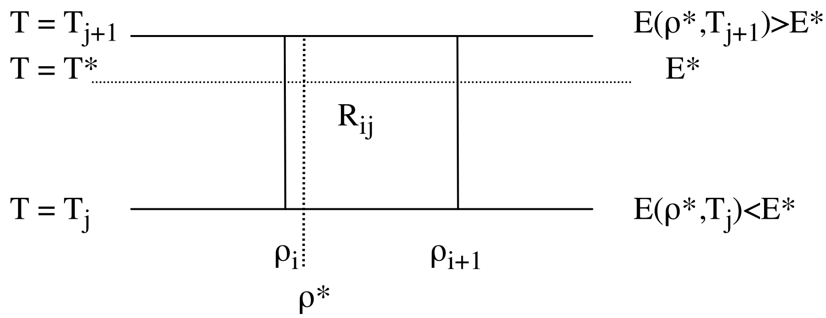
1.8. Inverse interpolation

Many equation of state (EOS) applications use ρ and E as the fundamental variables. In order to use EOS tables, which use (ρ,T) as independent variables, it is necessary to do an inverse lookup. Given values $(\rho,E) = (\rho^*,E^*)$, we need to find a temperature $T=T^*$ such that

$$E(\rho^*,T^*) = E^*. \tag{1.5}$$

The value (ρ^*,T^*) can then be used as (x_{int},y_{int}) in the standard evaluation procedure. This was the original motivation for including an inversion feature in LIP. At present, inversion is only supported in the second independent variable (called T here).

The same lookup procedure described above is used to find i such that $\rho_i \le \rho^* < \rho_{i+1}$. It is then necessary to examine the values $E_j = E(\rho^*,T_j)$, j = 0, 1, ..., ny–1 to find a j such that $E_j \le E^* < E_{j+1}$. A sequential search might require ny evaluations of E to determine j. To reduce code complexity, a simple binary search (without guess) is currently employed. Caution: This assumes that E is monotonic in T; i.e., $E_j < E_{j+1}$, j = 0, ..., ny–2.



We observe that these evaluations are also made simpler by using the fact that $T=T_j$ implies that y=0, and $T=T_{j+1}$ implies that y=1. (But not in the biquadratic case, as discussed in Section 5.3, below.)

At version 1.2, `lip_inverse_vals` was reorganized to support partial setup options. When the full coefficient array is available, `lip_inv_coeff` proceeds as above, using formulas that depend directly on the functional form. At the opposite extreme is `lip_inv_general`, which implements a modified Newton-Raphson iteration and computes new $E(\rho^*,T_j)$ values as needed by calls to `lip_evalu_bivar`. Intermediate between these is `lip_inv_partial`, which computes coefficients for each new box (i, j´) encountered in the above binary search from available information, depending on the setup mode. The new argument `inv_mode` was added to `lip_inverse_vals` (thus introducing an incompatibility with version 1.1) to control usage of these three sub-functions.

Note: In order to avoid complications due to round-off error, `lip_inv_general` modifies the above procedure so that if E is within `THRESH` of a box boundary, a two-interval bracket is passed to the Newton-Raphson iteration. (Currently, `THRESH` is 0.1.)

If the input ρ-value is out of range, the above-mentioned inversion is carried out using the appropriate border strip. If the E-value is out of range, the algorithm returns the associated boundary T-value, with zero T-derivative. (There is no linear extrapolation option.)

Note that the current C code uses generic variable names x, y, f rather than the ρ, T, E used above.

2. Piecewise Bilinear Interpolation

This section describes the bilinear interpolation method as currently supported by LIP.

2.1. The bilinear form

The bilinear functional form on the mesh rectangle R_{ij} is:

$$f(x_{int},y_{int}) = l(x,y) = a_0 + a_1x + a_2y + a_3xy \ , \tag{2.1}$$

where (x,y) are as in (1.3). Once the coefficients a_k have been determined for a given mesh rectangle R_{ij}, it is straightforward to evaluate the bilinear form at any (x_{int},y_{int}) in the box from (2.1). See end of Section 1.1 for notation used below.

2.2. Calculating interpolation coefficients

The interpolation conditions on mesh rectangle R_{ij} are:

$$f_{mn} = f(x_m,y_n), \ \ m = imin, imax, \ \ n = jmin, jmax. \tag{2.2}$$

There are four coefficients in (2.1) and four conditions in (2.2). Using (2.2), we can write down the coefficients immediately from (2.1).

$$f_{imin,jmin} \ = \ f(x_{imin},y_{jmin}) \ = \ l(0,0) \ = \ a_0; \tag{2.3a}$$

$$f_{imax,jmin} \ = \ f(x_{imax},y_{jmin}) \ = \ l(1,0) \ = \ a_0 + a_1; \tag{2.3b}$$

$$f_{imin,jmax} \ = \ f(x_{imin},y_{jmax}) \ = \ l(0,1) \ = \ a_0 + a_2; \tag{2.3c}$$

$$f_{imax,jmax} \ = \ f(x_{imax},y_{jmax}) \ = \ l(1,1) \ = \ a_0 + a_1 + a_2 + a_3. \tag{2.3d}$$

(2.3a) gives us a_0 directly:

$$a_0 = f_{imin,jmin}. \tag{2.4a}$$

From (2.3b) and (2.4a) we have

$$a_1 = f_{imax,jmin} - f_{imin,jmin}. \tag{2.4b}$$

Similarly, (2.3c) and (2.4a) yield

$$a_2 = f_{imin,jmax} - f_{imin,jmin}. \tag{2.4c}$$

Finally, (2.3d) gives us

$$a_3 = f_{imax,jmax} - (a_0 + a_1 + a_2). \tag{2.4d}$$

We observe that, by construction, the bilinear form will be continuous across the box boundaries. However, derivatives will have jump discontinuities there.

The interpolation coefficients are laid out in blocks of four in memory, with the coefficients for mesh rectangle R_{ij}, the one with lower-left corner at (xlow,ylow), starting at location `(j*(nx-1)+i)*4` in the coefficient array.

2.3. Direct inversion (bilinear)

Once the appropriate T-interval has been found, as discussed in Section 1.8, we need to solve equation (1.5) for T=T*. In the bilinear case, this is quite simple. From (2.1) we have

$$E(\rho^*,T) = a_0 + a_1x^* + a_2y + a_3x^*y, \tag{2.5}$$

where

$$x^* = (\rho^* - \rho_{imin}) / (\rho_{imax} - \rho_{imin}).$$

Equation (2.5) can be solved directly for y,

$$y = (E^* - (a_0 + a_1x^*)) / (a_2 + a_3x^*), \tag{2.6}$$

and the desired value (obtained by inverting (1.3y) with $y_{int} = T$) is:

$$T^* = (T_{jmax} - T_{jmin})\, y + T_{jmin}. \tag{2.7}$$

2.4. The univariate analog

The linear functional form on the mesh interval $[x_i, x_{i+1}]$ is:

$$f(x_{int}) \;=\; l(x) \;=\; a_0 + a_1x, \tag{2.8}$$

where, unlike in (1.3x), we do *not* divide by the interval length:

$$x \;=\; \mathrm{xscale}(x_{int}) \;=\; x_{int} - x_{imin}. \tag{2.9}$$

Since a univariate linear function is uniquely determined by its values at two distinct points, it is easy to compute the linear interpolation coefficients. The linear interpolant on the mesh interval $[x_i, x_{i+1}]$ is thus determined by

$$f_k = f(x_k),\;\; k = \;\mathrm{imin}, \mathrm{imax}. \tag{2.10}$$

Using (2.8) and (2.9), we can write conditions for the coefficients:

$$f_{imin} \;=\; f(x_{imin}) \;\;=\; l(0) \qquad\qquad = a_0; \tag{2.11a}$$

$$f_{imax} \;=\; f(x_{imax}) \;\;=\; l(x_{imax} - x_{imin}) \;\;= a_0 + a_1(x_{imax} - x_{imin}). \tag{2.11b}$$

(2.11a) gives us a_0 directly:

$$a_0 = f_{imin}. \tag{2.12a}$$

From (2.11b) and (2.12a) we have

$$a_1 = (f_{imax} - f_{imin}) / (x_{imax} - x_{imin}). \tag{2.12b}$$

The same formulas are used, with the obvious change of notation, if the independent variable is y instead of x.

3. Piecewise Bicubic Interpolation

This section describes the reduced bicubic interpolation method supported by LIP. (The term “reduced” refers to the fact that it only has 12 terms, rather than the 16 in the “full” bicubic)

3.1. The bicubic form

The “LIP standard” bicubic functional form on the mesh rectangle R_{ij} is:

$$\begin{aligned} f(x_{int},y_{int}) = c(x,y) = {} & a_0 + a_1x + a_2y + a_3xy + \\ & a_4x^2 + a_5x^2y + a_6x^3 + a_7x^3y + \\ & a_8y^2 + a_9xy^2 + a_{10}y^3 + a_{11}xy^3, \end{aligned} \tag{3.1}$$

where the four highest-order terms (x^2y^2, x^3y^2, x^2y^3, x^3y^3) have been omitted from the general bicubic polynomial to reduce storage space and evaluation time. Because the first four terms in (3.1) are the same as in (2.1), we note that a bilinear function can be viewed as a bicubic with its last eight coefficients equal to zero. See end of Section 1.1 for notation used below.

3.2. Calculating interpolation coefficients

The same four interpolation conditions (2.2) apply to the bicubic form on mesh rectangle R_{ij}, but there are twelve coefficients in (3.1), so we must find eight additional equations. If we had the values of the first partial derivatives of f at the data points, we could supplement (2.2) with the eight derivative interpolation conditions:

$$D_xf_{mn} = \partial f(x_m,y_n)/\partial x, \quad m = imin, imax, \; n = jmin, jmax. \tag{3.2a}$$

$$D_yf_{mn} = \partial f(x_m,y_n)/\partial y, \quad m = imin, imax, \; n = jmin, jmax. \tag{3.2b}$$

where D_xf_{mn} is the x-derivative of f at the m,n data point, and similarly for D_yf_{mn}. To approximate the needed derivatives, we bring in information from neighboring points. See Section 3.2.1, below, for details on how this is done.

Requiring the derivatives of bicubic (3.1) to match the two values $D_xf_{imin,jmin}$ and $D_xf_{imax,jmin}$ in (3.2a) gives two additional equations. Applying this procedure at $y = y_{jmax}$ gives two more. The other four equations are determined similarly, by reversing the roles of x and y and using (3.2b) instead of (3.2a).

Differentiating (3.1), and taking (1.3) into account, gives the partial derivatives required for (3.2a) and (3.2b) in R_{ij}:

$$\begin{aligned} \partial f(x_{int},y_{int})/\partial x = \partial c(x,y)/\partial x / \Delta x_i = [\, & a_1 + a_3y + 2a_4x + 2a_5xy + \\ & 3a_6x^2 + 3a_7x^2y + a_9y^2 + a_{11}y^3] / \Delta x_i. \end{aligned} \tag{3.3a}$$

$$\begin{aligned} \partial f(x_{int},y_{int})/\partial y = \partial c(x,y)/\partial y / \Delta y_j = [\, & a_2 + a_3x + a_5x^2 + a_7x^3 + \\ & 2a_8y + 2a_9xy + 3a_{10}y^2 + 3a_{11}xy^2] / \Delta y_j. \end{aligned} \tag{3.3b}$$

This leads to a system of 12 linear equations to be solved for the 12 coefficients (a_0, a_1, …, a_{11}) in (3.1). Note that properties (1.4) greatly simplify the matrix setup. In fact, the matrix is constant, independent of the data. It is set up for interpolation on the unit square with the first four rows of the matrix containing the conditions for interpolating the data at the four corners, f(`x[i]`,`y[i]`) = `rhs[i]`, where

```
x[ 0] = 0;  y[ 0] = 0;
x[ 1] = 0;  y[ 1] = 1;
x[ 2] = 1;  y[ 2] = 0;
x[ 3] = 1;  y[ 3] = 1.
```

Rows 4–7 contain x-derivative interpolation conditions, ∂f(`x[i]`,`y[i]`)/∂x = `rhs[i]`, where

```
x[ 4] = 0;  y[ 4] = 0;
x[ 5] = 1;  y[ 5] = 0;
x[ 6] = 0;  y[ 6] = 1;
x[ 7] = 1;  y[ 7] = 1.
```

Rows 8–11 contain y-derivative interpolation conditions, ∂f(`x[i]`,`y[i]`)/∂y = `rhs[i]`, where

```
x[ 8] = 0;  y[ 8] = 0;
x[ 9] = 0;  y[ 9] = 1;
x[10] = 1;  y[10] = 0;
x[11] = 1;  y[11] = 1.
```

Note that the ∂f/∂x conditions are not in the same order as the others. This is because it is natural to generate x-derivative estimates with y constant, y-derivative estimates with x constant.

The resulting constant matrix is (in C notation):

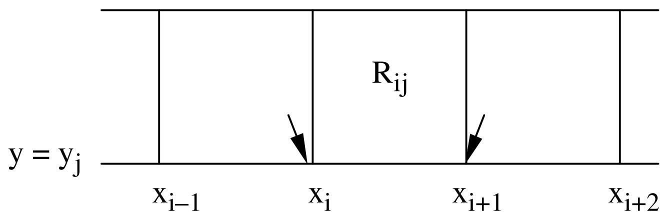
```
int imat[] = { 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
               1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, \
               1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, \
               1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, \
               0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
               0, 1, 0, 0, 2, 0, 3, 0, 0, 0, 0, 0, \
               0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, \
               0, 1, 0, 1, 2, 2, 3, 3, 0, 1, 0, 1, \
               0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
               0, 0, 1, 0, 0, 0, 0, 0, 2, 0, 3, 0, \
               0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, \
               0, 0, 1, 1, 0, 1, 0, 1, 2, 2, 3, 3  };     (3.4)
```

The twelve coefficients in (3.1) are computed by solving a 12×12 linear system, with the above matrix, using Gaussian elimination with partial pivoting. The procedure is to perform an LU factorization of the matrix exactly once. Then all of the linear systems solves reduce to a much more rapid back-substitution.

The interpolation coefficients are laid out in blocks of 12 in memory, with the coefficients for mesh rectangle R_{ij}, the one with lower-left corner at (xlow,ylow), starting at location `(j*(nx-1)+i)*12` in the coefficient array.

3.2.1. Derivative approximation

First observe that the four points $(x, f(x,y_j)))$, $x = x_{i-1}, x_i, x_{i+1}, x_{i+2}$ determine a (univariate) cubic in x. One could evaluate the derivative of this cubic at x_i and x_{i+1} to provide estimates of $D_x f_{ij}$ and $D_x f_{i+1,j}$ in (3.2a), and that was done in version 1.1.



A second derivative estimate at x_i can be obtained by using the above procedure with the cubic determined by the four points $(x, f(x,y_j)))$, $x = x_{i-2}, x_{i-1}, x_i, x_{i+1}$. To provide for continuity across mesh boundaries, the average of these two derivative estimates is used to approximate $D_x f_{ij}$. Similarly, the average of the two estimates from $x_{i-1}, x_i, x_{i+1}, x_{i+2}$ and $x_i, x_{i+1}, x_{i+2}, x_{i+3}$ is used to approximate $D_x f_{i+1,j}$. (A similar procedure with the roles of x and y interchanged is used to provide y-derivative estimates.)

Because the necessary neighboring values needed to determine the univariate cubics are not available in the boundary boxes, a non-centered four-point estimate is used in these cases, to retain cubic precision. This value is set to zero of it is of the opposite sign from the data slope at the boundary.

This is the procedure, implemented in `lip_cubic_derivs`, used all along for the derivative estimates needed for bicubic Hermite coefficients, Section 4.2, below. This averaging process was not carried out for the standard LIP bicubic in version 1.0, and the estimate dropped to quadratic in the normal direction in boundary boxes. Because a different set of four data points was used to estimate $D_x f_{ij}$ in box (i–1,j) and box (i,j), $\partial f/\partial x$ might not even be continuous at $x = x_i$ along the mesh lines. (Similar remarks apply to $\partial f/\partial y$.) At version 1.1, to simplify support of partial setup, it was decided to use the same procedure in both cases, even though this meant that values computed by the standard bicubic differed between the two versions. [*Caution*: Function `lip_cubic_derivs` *does* assume the independent variable array is increasing. (Not checked there.)]

In the standard 12-term bicubic case, it turns out that derivatives are still not quite continuous. This is presumably due to the fact that we are not using a complete 16-term bicubic here. Note that the bicubic Hermite interpolant *does* have continuous function and first partial derivatives.

3.3. Modification for two-phase data

A modified version of the bicubic coefficient setup is used for two-phase data, because the standard bicubic behaves very badly at the edges of the two-phase region, where the data suddenly changes from being constant in x (density) to changing very rapidly in this variable. The procedure is to drop to bilinear (a special case of bicubic) inside or at the boundary of the two-phase region. Note: With the advent of bimond (see Section 4.3), *this method is considered deprecated*.

The two-phase region is detected by the test in `lip_flat_for_2p` (in C notation):

```
if (
 (i > 0 && \
  isflat(data[ j   *nx + i-1], data[ j   *nx + i  ])) || \

  isflat(data[ j   *nx + i  ], data[ j   *nx + i+1])  || \

 (i < nx-2 && \
  isflat(data[ j   *nx + i+1], data[ j   *nx + i+2])) || \

 (i > 0 && \
  isflat(data[(j+1)*nx + i-1], data[(j+1)*nx + i  ])) || \

  isflat(data[(j+1)*nx + i  ], data[(j+1)*nx + i+1])  || \

 (i < nx-2 && \
  isflat(data[(j+1)*nx + i+1], data[(j+1)*nx + i+2]))   )
{
  goto Make_it_bilinear;
}
```
(3.5)

Here the logical function `isflat` is defined by

$$\texttt{isflat}(a, b) = (|a-b| / \max(|a|, |b|) \leq \texttt{FLATHRSH}), \tag{3.6}$$

where the "flatness threshold" `FLATHRSH` is a code parameter that is currently equal to `1.0e-7`. To reduce the amount of extra testing, (3.5) is applied only in boxes R_{ij} which satisfy

$$x_i < 1.5\ x_{crit},\quad y_j \leq 1.1\ y_{crit}.$$

For a typical EOS application, $x_{crit}=\rho_0$, the "normal density" for the material, and $y_{crit}=T_c$, the "critical temperature".

In order to reduce discontinuities between bicubic boxes and neighboring bilinear boxes, the derivative estimates at neighboring points are modified to match the linear slopes.

Note that the above procedure was intended primarily to handle the characteristics of two-phase pressure data.

3.4. Inverse iteration

Once the appropriate T-interval has been found, as in Section 1.8, we need to solve equation (1.5) for T=T*. The bicubic case is much more complicated than the bilinear case. In this case we have to solve a cubic polynomial equation for y. The present code uses a hybrid secant/bisection algorithm to solve this. Matters are simplified a bit by the fact that, due to the variable transformation (1.3y), we are solving for a root in the interval [0,1].

The iteration tolerance `NEWTON_TOL` is currently set at 1.0e-7. (This was 1.0e-5 in an earlier version, but that was deemed to be insufficient accuracy.) A typical call requires 5–7 iterations, but both smaller and larger values have been observed. (An earlier version that used bisection exclusively required 15 iterations per call.)

3.5. The univariate analog

The cubic functional form on the mesh interval $[x_i, x_{i+1}]$ is:

$$f(x_{int}) = c(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3, \tag{3.7}$$

where x is as in (2.9).

A univariate cubic function is uniquely determined by the values of the function and its first derivative at two distinct points. The cubic interpolant on the mesh interval $[x_i, x_{i+1}]$ is thus determined by (2.10) and

$$d_k = f'(x_k),\ k = \text{imin}, \text{imax}. \tag{3.8}$$

Differentiating (3.7) gives

$$f'(x_{int}) = c'(x) = a_1 + 2a_2 x + 3a_3 x^2 , \tag{3.9}$$

so that we have the four conditions:

$$f_{imin} = f(x_{imin}) = c(0) = a_0; \tag{3.10a}$$

$$d_{imin} = f'(x_{imin}) = c'(0) = a_1; \tag{3.10b}$$

$$f_{imax} = f(x_{imax}) = c(x_{imax}-x_{imin}) = a_0 + a_1\Delta x_i + a_2\Delta x_i^2 + a_3\Delta x_i^3; \tag{3.10c}$$

$$d_{imax} = f'(x_{imax}) = c'(x_{imax}-x_{imin}) = a_1 + 2a_2\Delta x_i + 3a_3\Delta x_i^2, \tag{3.10d}$$

where $\Delta x_i = x_{imax}-x_{imin}$.

(3.10a) and (3.10b) give us a_0 and a_1 directly:

$$a_0 = f_{imin}; \tag{3.11a}$$

$$a_1 = d_{imin}. \tag{3.11b}$$

Substituting these into (3.10c) and (3.10d) yields a pair of equations to be solved for the remaining two coefficients. From these we obtain:

$$a_2 = -(2\boldsymbol{\Delta}_{imin} + \boldsymbol{\Delta}_{imax}) ; \tag{3.11c}$$

$$a_3 = (\boldsymbol{\Delta}_{imin} + \boldsymbol{\Delta}_{imax}) / (x_{imax}-x_{imin}) , \tag{3.11d}$$

where

$$\Delta_k = (d_k - m) / (x_{imax} - x_{imin}), \; k = \text{imin, imax} ,$$

and m is the data slope, $m = (f_{i+1} - f_i) / (x_{i+1} - x_i)$.

If we use the same derivative estimation scheme to produce d_{imin} and d_{imax} as is used for $D_x f_{mn}$ in the bicubic setup, we obtain a univariate cubic interpolant that is compatible with the bivariate bicubic, in the sense discussed in Section 1.2, above.

As in the bilinear case, the same formulas are used with the obvious change of notation if the independent variable is y instead of x.

4. Bicubic Hermite Interpolation (biherm)

This section describes the bicubic Hermite interpolation method (abbreviated "biherm") as currently supported by LIP.

4.1. The bicubic Hermite form

The bicubic Hermite functional form on the mesh rectangle R_{ij} is:

$$\begin{aligned} f(x_{int}, y_{int}) \;=\; b(x,y) \;=\; & \\ & a_0\, h_0(x)h_0(y) + a_1\, h_1(x)h_0(y) + a_2\, h_2(x)h_0(y) + a_3\, h_3(x)h_0(y) + \\ & a_4\, h_0(x)h_1(y) + a_5\, h_1(x)h_1(y) + a_6\, h_2(x)h_1(y) + a_7\, h_3(x)h_1(y) + \\ & a_8\, h_0(x)h_2(y) + a_9\, h_1(x)h_2(y) + a_{10} h_2(x)h_2(y) + a_{11} h_3(x)h_2(y) + \\ & a_{12} h_0(x)h_3(y) + a_{13} h_1(x)h_3(y) + a_{14} h_2(x)h_3(y) + a_{15} h_3(x)h_3(y) \,. \end{aligned} \tag{4.1}$$

For improved numerical stability, we have performed a change of basis on the bicubic form. Note that this is not equivalent to (3.1), because it contains the full 16 terms required for a general bicubic function. Note also that a bilinear function is *not* a special case.

The (univariate) *cubic Hermite basis functions* that appear in (4.1) are defined by relations:

$$h_0(0) = 1, \quad h_0(1) = 0, \quad h_0'(0) = 0, \quad h_0'(1) = 0; \tag{4.2a}$$
$$h_1(0) = 0, \quad h_1(1) = 1, \quad h_1'(0) = 0, \quad h_1'(1) = 0; \tag{4.2b}$$
$$h_2(0) = 0, \quad h_2(1) = 0, \quad h_2'(0) = 1, \quad h_2'(1) = 0; \tag{4.2c}$$
$$h_3(0) = 0, \quad h_3(1) = 0, \quad h_3'(0) = 0, \quad h_3'(1) = 1. \tag{4.2d}$$

These lead to the following formulas for the basis functions:

$$h_0(t) \;=\; 1 - 3\,t^2 + 2\,t^3 \;=\; h_1(1-t) \;=\; u^2\,(u + 3\,t)\,; \tag{4.3a}$$
$$h_1(t) \;=\; 3\,t^2 - 2\,t^3 \;=\; t^2\,(t + 3\,u)\,; \tag{4.3b}$$
$$h_2(t) \;=\; t - 2\,t^2 + t^3 \;=\; -h_3(1-t) \;=\; u^2\,t\,; \tag{4.3c}$$
$$h_3(t) \;=\; -\,t^2 + t^3 \;=\; -t^2\,u\,, \tag{4.3d}$$

where we have set $u = 1 - t$.

We note that equation (4.1) can be rewritten in matrix notation:

$$b(x,y) \;=\; \mathbf{h}(y)^T\, \mathbf{A}\, \mathbf{h}(x) \;, \tag{4.4a}$$

where

$$\mathbf{h}(t) = (\, h_0(t), h_1(t), h_2(t), h_3(t)\,)^T \;, \tag{4.4b}$$

and

$$\mathbf{A} \;=\; \begin{pmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \\ a_{12} & a_{13} & a_{14} & a_{15} \end{pmatrix} . \tag{4.4c}$$

For evaluation purposes, formula (4.4a) can be associated either from the left or right, for two possible nested four-element summations. In one case linear combinations of the x-basis functions are formed, and the results are used to form linear combinations of the y-basis functions. The reverse is the case if the other association is chosen. After some experimentation, different associations have been used in the evaluator, depending on the evaluation history, in an attempt to minimize evaluation time. The result is that a biherm evaluation is only 20 to 40 percent slower than a standard bicubic one (depending on whether derivatives are evaluated).

Differentiating (4.4a) with respect to x yields:

$$\partial b(x,y)/\partial x \;=\; \mathbf{h}(y)^T\, \mathbf{A}\, \mathbf{h}'(x) \;, \tag{4.5}$$

Similarly,

$$\partial b(x,y)/\partial y \;=\; \mathbf{h}'(y)^T\, \mathbf{A}\, \mathbf{h}(x) \;, \tag{4.6}$$

For bicubic Hermite derivative evaluation, formula (4.5) or (4.6) is used. To provide the necessary values, the Hermite basis derivative evaluator returns **h**´. The relevant formulas are obtained by differentiating (4.3), namely:

$$h_0'(t) \;=\; -6\,t + 6\,t^2 \;=\; -h_1'(1-t) \;=\; -6tu \;; \tag{4.7a}$$
$$h_1'(t) \;=\; 6\,t - 6\,t^2 \;=\; 6tu \;; \tag{4.7b}$$
$$h_3'(t) \;=\; 1 - 4\,t + 3\,t^2 \;=\; h_4'(1-t) \;= u\,(u - 2\,t) \;; \tag{4.7c}$$
$$h_4'(t) \;=\; -2\,t + 3\,t^2 \;= t\,(t - 2\,u) \;. \tag{4.7d}$$

4.2. Calculating interpolation coefficients

The bicubic function b(x,y) on mesh box R_{ij}, after the transformations (1.3), is uniquely determined by the values of the four quantities

$$b,\; \partial b/\partial x,\; \partial b/\partial y,\; \partial^2 b/\partial x \partial y$$

at the corners of the box. By construction, if the same values of these four quantities are used in adjacent boxes, then these functions are continuous across the box boundaries. This requires using continuous derivative estimates for all partial derivatives that appear here if actual derivative values are unavailable. See end of Section 1.1 for notation used below.

4. Bicubic Hermite Interpolation (biherm)

A significant advantage of using the Hermite basis is that the interpolation coefficients can be established directly from the interpolation conditions without the need to solve any linear systems. For example, using the relations (4.2) in the functional form (4.1) immediately yields the interpolation conditions (2.2) and the following coefficients:

$$a_0 = b(0,0) = f(x_{imin},y_{jmin}) = f_{imin,jmin}\ ; \tag{4.8a}$$
$$a_1 = b(1,0) = f(x_{imax},y_{jmin}) = f_{imax,jmin}\ ; \tag{4.8b}$$
$$a_4 = b(0,1) = f(x_{imin},y_{jmax}) = f_{imin,jmax}\ ; \tag{4.8c}$$
$$a_5 = b(1,1) = f(x_{imax},y_{jmax}) = f_{imax,jmax}\ . \tag{4.8d}$$

Note that these four coefficients are in the upper left 4×4 corner of the coefficient matrix A in (4.4c).

Again applying (4.2) to the derivative interpolation conditions (3.2a) shows that the upper right corner of **A** contains the ∂b/∂x–values:

$$a_2 = \partial b(0,0)/\partial x = \Delta x_i\ \partial f(x_{imin},y_{jmin})/\partial x = \Delta x_i\ D_x f_{imin,jmin}\ ; \tag{4.9a}$$
$$a_3 = \partial b(1,0)/\partial x = \Delta x_i\ \partial f(x_{imax},y_{jmin})/\partial x = \Delta x_i\ D_x f_{imax,jmin}\ ; \tag{4.9b}$$
$$a_6 = \partial b(0,1)/\partial x = \Delta x_i\ \partial f(x_{imin},y_{jmax})/\partial x = \Delta x_i\ D_x f_{imin,jmax}\ ; \tag{4.9c}$$
$$a_7 = \partial b(1,1)/\partial x = \Delta x_i\ \partial f(x_{imax},y_{jmax})/\partial x = \Delta x_i\ D_x f_{imax,jmax}\ , \tag{4.9d}$$

where $\Delta x_i = x_{imax} - x_{imin}$.

Similarly, (3.2b) indicates the lower left corner has the ∂b/∂y–values:

$$a_8 = \partial b(0,0)/\partial y = \Delta y_j\ \partial f(x_{imin},y_{jmin})/\partial y = \Delta y_j\ D_y f_{imin,jmin}\ ; \tag{4.10a}$$
$$a_9 = \partial b(1,0)/\partial y = \Delta y_j\ \partial f(x_{imax},y_{jmin})/\partial y = \Delta y_j\ D_y f_{imax,jmin}\ ; \tag{4.10b}$$
$$a_{12} = \partial b(0,1)/\partial y = \Delta y_j\ \partial f(x_{imin},y_{jmax})/\partial y = \Delta y_j\ D_y f_{imin,jmax}\ ; \tag{4.10c}$$
$$a_{13} = \partial b(1,1)/\partial y = \Delta y_j\ \partial f(x_{imax},y_{jmax})/\partial y = \Delta y_j\ D_y f_{imax,jmax}\ , \tag{4.10d}$$

where $\Delta y_j = y_{jmax} - y_{jmin}$.

See Section 3.2.1, above, for a description of the procedure used to provide estimates for the first partial derivatives in (4.9) and (4.10). By using continuous estimates, this assures continuity of first derivatives of the interpolant.

The mixed partial derivatives (or "twists") fill out the remainder of the coefficient matrix. To see this, differentiate either (4.5) or (4.6) with respect to the other variable to obtain:

$$\partial^2 b(x,y)/\partial x \partial y = \mathbf{h}'(y)^T\ \mathbf{A}\ \mathbf{h}'(x)\ , \tag{4.11}$$

and the remaining coefficients:

$$a_{10} = \partial^2 b(0,0)/\partial x\partial y = \Delta x_i\ \Delta y_j\ \partial^2 f(x_{imin},y_{jmin})/\partial x\partial y\ ; \tag{4.12a}$$
$$a_{11} = \partial^2 b(1,0)/\partial x\partial y = \Delta x_i\ \Delta y_j\ \partial^2 f(x_{imax},y_{jmin})/\partial x\partial y\ ; \tag{4.12b}$$
$$a_{14} = \partial^2 b(0,1)/\partial x\partial y = \Delta x_i\ \Delta y_j\ \partial^2 f(x_{imin},y_{jmax})/\partial x\partial y\ ; \tag{4.12c}$$
$$a_{15} = \partial^2 b(1,1)/\partial x\partial y = \Delta x_i\ \Delta y_j\ \partial^2 f(x_{imax},y_{jmax})/\partial x\partial y\ . \tag{4.12d}$$

Note that the twists could be set to zero without losing the continuity properties, thus reducing the storage requirements to that of the bicubic form, in exchange for a reduction in accuracy. We have not chosen this option for LIP. Instead, the average of the three-point difference formulas in the two coordinate directions applied to the current first derivative estimates is used to estimate the twists in (4.12).

These formulas are used by the biherm setup routine and result in an interpolant that is exact when interpolating data from a biquadratic function. While higher-order twist estimates could be used to obtain complete bicubic precision, it has been decided that it is not worth the extra effort for such a minimal effect on the interpolant.

The interpolation coefficients are laid out in blocks of 16 in memory, with the coefficients for mesh rectangle R_{ij}, the one with lower-left corner at (xlow,ylow), starting at location `(j*(nx-1)+i)*16` in the coefficient array.

Note that, since bilinear functions are not a subset of bicubic Hermite functions, there is no convenient way to drop to bilinear inside the two-phase region. Consequently, there is no modification of the biherm interpolant for two-phase data comparable to that discussed in Section 3.3. Experience has shown that this interpolant does “ring” near the phase transition boundary, but the behavior is confined to only boxes adjacent to this boundary and is not nearly as pathological as the standard bicubic. To eliminate this “ringing” altogether, use the bimond interpolant, described in the next section.

4.3. Monotone bicubic Hermite (bimond)

Monotonicity preservation is possible with the Hermite form of the bicubic interpolant (see [2]–[4]). We have extended the algorithm described in [4] to handle piecewise monotonic data (such as a typical pressure table) and included it in LIP as the bimond option.

The name “bimond” is historical. The original version of our univariate monotone piecewise cubic interpolation algorithm was implemented in subroutine `MONDER` (`MON`otone `DER`ivatives), a misnomer for the fact that it determined derivative values that resulted in a monotone piecewise cubic Hermite interpolant. We have retained this name for the univariate routine described in Section 4.5, below. Quite naturally, the bivariate version became known as `BIMOND`. The incarnation included in LIP is `BIMOND5` (the fifth release).

In brief outline, the `BIMOND5` algorithm proceeds as follows:

- Step 1. Initialize two arrays that characterize the monotonicity properties of the data. (In each segment where the data have a common monotonicity sense, the interpolant will preserve that monotonicity, except perhaps in boxes adjacent to a switch in data monotonicity).
- Step 2. Compute initial values for the first partial derivatives $\partial b/\partial x$ and $\partial b/\partial y$ that satisfy a sufficient condition for monotonicity along the mesh lines. This is done by first initializing these values as described above for the biherm option, and

then screening them, possibly reducing derivative magnitudes to satisfy the condition.

Step 3. Construct intervals of acceptable values, containing zero, for the twists $\partial^2 b/\partial x \partial y$. This step may require further reduction in magnitude of first partial derivatives.

Step 4. Compute values for the twists, as described above for the biherm option, and map them into the intervals determined in Step 3.

The primary complication beyond `BIMOND4` [4] occurs in detecting boundaries of monotonicity regions and treating derivative values in adjacent boxes appropriately.

Once the interpolation coefficients have been determined via the `BIMOND5` algorithm, the resulting function is evaluated and/or inverted in exactly the same way as an ordinary biherm interpolant. Only the coefficient setup is different.

The situation becomes rather more complicated in partial setup mode. In mode (2), the `BIMOND5` derivative adjustments can be done once and for all at setup time, so bimond is no different than biherm at evaluation time. The complication comes when only data is available. Depending on the function values, the derivative adjustments may require access to the entire data set. This would require doing the equivalent of a full bimond setup, with its attendant large temporary memory allocation and execution time, to compute coefficients for each new mesh box encountered during the evaluation phase. To reduce the needed storage and time, `lip_evalu_bimond` was modified at version 1.2 to only consider a subset of the data array `nxexp` points in the x-direction and `nyexp` in the y-direction on either side if the current mesh box when computing coefficients. These "expansion parameters" are global variables in `lip_eval2D.c`. The default values are `nxexp=nyexp=5`, but the user may reset them by calling `lip_set_bimond_exp` prior to calling `lip_evalu_bivar`. The larger these parameters are, the more temporary storage and execution time is required, but the more likely the results will agree with those from full coefficient setup mode. (Note that the default values were 3 prior to Version 1.4.1)

4.4. Inverse iteration

The bicubic Hermite case is handled much like the standard bicubic case, discussed in Section 3.4, above. The same iteration procedure is used. The only difference is that the cubic being solved is represented in Hermite form, rather than power form.

The inversion of a bimond interpolant in data only mode requires the computation of interpolation coefficients for each mesh box examined. Because of the above complication, the use of `lip_inv_general` (see Section 1.8) requires the repeated allocation and de-allocation of storage and expensive coefficient calculation for each new j-index considered. The algorithm employed in this case by `lip_inv_partial` computes coefficients for the whole strip of boxes above the current i-interval before entering the search for the appropriate j-index. (There is, in fact a provision for providing limits `jmin`, `jmax` for this strip, but that is not generally activated in version 1.5.1, except in the special case of a single inversion point.) This requires more temporary storage to be allocated, but saves execution time. To take maximum advantage of this feature, the inversion points should be ordered so that all x-values (ρ-values, in the notation of Section 1.8) in the same mesh interval are grouped together.

4.5. The univariate analog

The cubic Hermite functional form on the mesh interval $[x_i, x_{i+1}]$ is:

$$f(x_{int}) = c(x) = a_0\, h_0(x) + a_1\, h_1(x) + a_2\, h_2(x) + a_3\, h_3(x), \tag{4.13}$$

where x is as in (1.3x), and the $h_k(x)$ are as in (4.3). (We *do* divide by the interval length here, because the h_k are defined on the unit interval [0,1].)

Differentiating (4.13) and using the defining characteristics of the cubic Hermite basis functions (4.2) yields the four formulas:

$$a_0 = c(0) = f(x_{imin}) = f_{imin}\,; \tag{4.14a}$$

$$a_1 = c'(0) = \Delta x_i\, f'(x_{imin}) = \Delta x_i\, d_{imin}\,; \tag{4.14b}$$

$$a_2 = c(1) = f(x_{imax}) = f_{imax}\,; \tag{4.14c}$$

$$a_3 = c'(1) = \Delta x_i\, f'(x_{imax}) = \Delta x_i\, d_{imax}\,, \tag{4.14d}$$

These immediately give us the fact that the cubic Hermite function (4.13) satisfies the conditions (2.10) and (3.8) that are necessary and sufficient to uniquely define a cubic function on $[x_{imin}, x_{imax}]$.

If we use the same derivative estimation scheme to produce d_{imin} and d_{imax} as is used for $D_x f_{mn}$ in the bicubic Hermite setup, then we obtain a univariate cubic Hermite interpolant that is compatible with its bivariate counterpart, in the sense discussed in Section 1.2, above.

In order to provide a univariate interpolant (monder) that is compatible with bimond, we have included an algorithm that uses essentially the procedure of Step 2 of `BIMOND5` (see Section 4.3, above) to compute a piecewise monotonic cubic Hermite interpolant. Strictly speaking, monder will be compatible with the bimond interpolant only of no Step 3 first derivative modifications were required along the lowest isotherm.

5. Piecewise Biquadratic Interpolation

This section describes the biquadratic interpolation method as currently supported by LIP.

5.1. The biquadratic form

The biquadratic form is *not* recommended as a general bivariate interpolator. It is included in LIP only as a means to provide an interpolant for the old EOP data that emulates that provided in EOS4 [5]. There is no transformation to the normalized variables (x,y) in this case; that is, $(x,y) = (x_{int},y_{int})$ here. This is to be as close to the EOS4 interpolant as possible (outside of language and computer arithmetic differences).

The biquadratic functional form on the mesh rectangle R_{ij} is:

$$f(x_{int},y_{int}) = q(x,y) = a_0 + a_1 x + a_2 y + a_3 x^2 + a_4 y^2 + a_5 xy + a_6 x^2 y + a_7 x y^2 + a_8 x^2 y^2. \quad (5.1)$$

NOTE: This form is not fully supported; in particular, there is no univariate equivalent.

5.2. Calculating interpolation coefficients

For compatibility with the past, the coefficient calculation algorithm used is a C translation of the one in EOS4 [5]. (Further details will not be given here.)

The interpolation coefficients are laid out in blocks of nine in memory, with the coefficients for mesh rectangle R_{ij} starting at location `(i*(nx-1)+j)*9` in the coefficient array.

5.3. Inverse iteration

The biquadratic case is intermediate in complication between bilinear and bicubic. In this case, a quadratic equation has to be solved, once we determine the interval $[T_i,T_{i+1}]$ containing the target T*. For compatibility with the past, the algorithm used is a C translation of the one in EOS4 [5], with some of the special case coding omitted.

6. Piecewise Birational Interpolation

This section describes the birational interpolation method, which was added to LIP at version 1.3.

6.1. The birational form

The birational form does not fit naturally into the standard LIP formulation. According to the available LANL documentation [7], the univariate rational form is the ratio of a cubic and a linear polynomial:

$$r(t) = [f_k + A_k (t - t_k) + B_k (t - t_k)^2 + D_k (t - t_k)^3] \ / \ [1+ C_k (t - t_k)], \quad (6.1)$$

for t in the k-th mesh interval, $[t_k, t_{k+1}]$. The coefficients that appear in (6.1) are not computed explicitly, but rather the function value is developed from the data values in the course of evaluation.

The birational form is obtained, in principle, by applying this procedure in each of the mesh directions. It is not guaranteed that the univariate rational interpolator is compatible in the sense of Section 1.2.

6.2. Evaluation

The code provided in LIP was created by adapting C code developed at LANL to LIP specifications. As noted in Section 7.8, below, this is contained in files `lip_BiRatInterp.c` and `lip_RatInterp.c`.

6.3. Inverse iteration

The only inversion procedure currently supported for the birational form is the use of `lip_inv_general` (see Section 1.8).

7. Software Organization

LIP is written in ANSI standard C for portability, but an object-oriented design is emulated in the code. The LIP source code is described in this section. First comes an overview illustrating how to use the software. An outline of the source code organization follows this. Then come detailed descriptions of the source files, grouped by function. At the end is a section on LIP programming conventions.

7.1. Overview

To use LIP to interpolate in a given data table, one must first create a LIP interpolation object, populate it with the data, and create a coefficient array for the desired interpolation method (unless a partial setup is desired). Then one can pass this object to other functions to do desired forward or inverse interpolations.

A summary of this process follows, with illustrative examples. For a more complete example of package usage, refer to Section 8, below.

Step (1): Create and populate an interpolation object with interpolation coefficients of the chosen interpolation type. (The following example uses pressure as a function of density and temperature, but it could be any two-dimensional data set. It uses bicubic Hermite interpolation.)

```
#include "LIP_proto.h"  /* LIP function prototypes. */
   ...
LIP_interp *interp ;
LIP_meth int_type=LIP_HERM ;
Integer job_flag, retval ;

job_flag = 3 ;   /* Do full coefficient setup. */

retval = lip_setup_interp ( &interp, "rho", ndens, dens,
                            "T", ntemp, temp, "P", pval,
                            int_type, job_flag );
```

Step (2): Pass `interp` on to functions that will do evaluations and/or inversions using it. (The following will interpolate at `npts` points, with the results in `fint`. Flag extr_flag has been set to select linear extrapolation, as discussed in Section 1.7. Only function values have been requested, so dfdx and dfdy will not be referenced.)

```
Integer extr_flag=1;
Logical calcf=TRUE, calcdfdx=FALSE, calcdfdy=FALSE;
retval = lip_evalu_bivar( interp, int_type, extr_flag,
                          xint, yint, npts, calcf, fint
                          calcdfdx, dfdx, calcdfdy, dfdy );
```

Step(3): When through using this object, free up space that has been allocated for it.

```
retval = lip_free_interp( &interp );
```

The user who is interested in the reduced storage requirements made available via the partial setup options discussed in Section 1.4, above, should refer to Section 7.3, below.

Aside: Those of you familiar with LIP version 1.0 will know that the process in Step (1) originally required several steps, each involving a call to one of the LIP setup functions:

(1) Create a LIP interpolation object (macro FMAKE).
(2) Initialize the object (function lip_init_interp).
(3) Add the data to it (function lip_add_data).
(4) Compute interpolation coefficients of the chosen interpolation type (function lip_calc_coeff).

A major change in version 1.1 was the introduction of a function lip_setup_interp that combines these steps into a single call. From the user's point of view, this reduces the amount of coding required to get started. Note that the object must now appear as a pointer to a pointer in the lip_setup_interp call.

Note ***incompatibility*** *with version 1.0*: While the old four-step process can still be used to set up the object, if desired, the lip_free_interp call sequence has changed, so the object must now appear as a pointer to a pointer. (Note that lip_free_interp now de-allocates the object, so an SFREE is not needed.)

7.2. LIP data types and memory management

In the above we referred to FMAKE and SFREE, which are memory management macros defined in LIP_macros.h. These were written to emulate the PACT [6] memory management facility, which was used in the original version of LIP. Their call sequences are:

FMAKE: Allocate space for a single object of type type.

```
obj = FMAKE( type, string );
```

Here obj is a pointer to the created object and string is an identifier, typically of the form "FNAME:obj", where FNAME is the name of the function issuing this call.

FMAKE_N: Allocate space for an array of n objects of type type.

```
obj= FMAKE_N( type, n, string );
```

Here obj is a pointer to the created array and string is an identifier, typically of the form “FNAME:obj”, where FNAME is the name of the function issuing this call. (See the sample program for examples.)

SFREE: De-allocate space for an object previously allocated by FMAKE or FMAKE_N:

```
SFREE( obj );
```

To support shared memory usage, a variant of FMAKE_N, called P_FMAKE_N, was introduced in version 1.6, with the same argument list. (The “P” stands for “persistent storage”.) The matching de-allocation should be done via P_SFREE. There are several other similar macros for future work, when the entire Interp_t instance will be allocated in shared memory. Currently, they do not do anything special, but are used to indicate where modifications may need to be done when this is implemented. They are intended for use only in LIP internal routines.

LIP_macros.h also contains definitions for min, max, and other useful macros.

All of the LIP source files also include LIP_Ftype.h, which defines Fortran-compatible data types such as Integer, Real8, Logical used above. Because header files LIP_Ftype.h and LIP_macros.h are included in LIP_proto.h, there is no need for the user to explicitly include them.

7.3. Partial setup options

In order to provide ready access to the full range of setup options available in LIP, we give below the complete user interface for lip_setup_interp. (This was taken from the source code and edited to better fit the page format.) Note that “partial setup” means job_flag < 3. Refer to Appendix A for definitions of the int_type and setup_type values mentioned here.

```
Integer lip_setup_interp( LIP_interp **interp,
                          const char *xname,
                          const Integer nx,
                          const Real8 *x,
                          const char *yname,
                          const Integer ny,
                          const Real8 *y,
                          const char *fname,
                          const Real8 *f,
                          const LIP_meth int_type,
                          const Integer job_flag )

/***********************************************************
This function initializes and populates a LIP interpolation
object.  Partial setup options are provided, as well as the
standard full coefficient array calculation.
```

```
Arguments:
   interp is a pointer to a newly created LIP_interp
          object.  Because its memory space is allocated
          within this function, this is a pointer to a
          pointer.  Thus the ADDRESS of the declared object
          should appear in the call list.  Example:
            LIP_interp *my_interp;
            retval = lip_setup_interp( &my_interp, xname,
                                       ...);
            < Code that passes my_interp to an evaluator,
              inverter, etc. >
          The allocated space will be freed by a call to
          lip_free_interp.
   xname  is the name to be associated with the x-values.
   nx     is the number of x-values.
   x      is a pointer to the array of x-values to be
          added.
   yname  is the name to be associated with the y-values.
   ny     is the number of y-values.
   y      is a pointer to the array of y-values to be
          added.
   fname  is the name to be associated with the f-values.
   f      is a pointer to the array of f-values to be
          added.
   int_type is the interpolation type for this object.
          When job_flag<3, an input value of 0 will be
          interpreted as "I don't want to set it yet,"
          which means it will have value LIP_INVALID(=0) on
          return.  This can be overridden by a subsequent
          lip_set_int_type call or by the int_type argument
          to lip_evalu_univar, lip_evalu_bivar, or lip_
          inverse_vals.
   job_flag indicates the type of setup to be performed.
          job_flag = 0 : only add the data to interp.
                   On successful completion, setup_type
                   will be LSU_DATA.
          job_flag = 1 : approximate first partial
                   derivatives, using a method consistent
                   with int_type.  (Not applicable if
                   int_type = LIP_LIN or LIP_RAT.  Acts
                   like job_flag=0 in this case.)
                   (Caution: While this option can be used
                   with int_type = LIP_MONO, this method
```

```
                   really requires twists for 2-D data.)
                   On successful completion, setup_type
                   will be LSU_1DER or LSU_2DER, depending
                   on whether 1-D or 2-D data was provided.
          job_flag = 2 : approximate first partial
                   derivatives and twists, using a method
                   consistent with int_type.  (Only appli-
                   cable for 2-D data with int_type =
                   LIP_HERM or LIP_MONO.  As above, if int_
                   type = LIP_LIN or LIP_RAT this acts like
                   job_flag=0; if int_type = LIP_CUB, it
                   acts like job_flag=1.)  On successful
                   completion, setup_type will be LSU_3DER.
         job_flag = 3 : compute full coefficient array for
                   int_type interpolation.  (Not applicable
                   if int_type = LIP_RAT. In this case, a
                   -1802 error is returned.)  On successful
                   completion, setup_type will be LSU_COEF.

  Note: If nx=0, x may be NULL and is not checked or added.
        If ny=0, y may be NULL and is not checked or added.
        (These are 1-D data sets.)
        If min(nx,ny)>0 (this is a 2-D data set), then f-
        values are assumed to be stored so that f[j*nx+i] =
        fcn(x[i],y[j]).

Input arguments:  xname, nx, x, yname, ny, y, fname, f,
                  int_type, job_flag.

Output arguments: interp.

Return value:  The return value, retval, should be zero.
  A positive return value is a warning, indicating that
  retval of the array fields in interp were non-NULL, so
  were freed before allocation.  (This probably indicates
  that lip_setup_interp was called with a previously
  populated interp.)
  The possible fatal error returns are:
    retval = -1800 : trouble creating interp.
    retval = -1801 : illegal value of job_flag.
    retval = -1802 : illegal or unsupported value of
                     int_type.
    retval = -1803 : too few data points for linear
```

```
                       interpolation.
    retval = -1804 : too few data points for rational
                       interpolation.
    retval = -1805 : too few data points for cubic
                       interpolation.
    retval = -1810 : trouble creating temporary workspace.
    retval = -1820 : error return from lip_monomod1D.
    retval = -1821 : error return from PBHpm.
  This version may also return retval from a called
  function.  In particular:
    retval = -1106 : One or both of the x, y arrays is not
                     strictly monotonic. (From lip_add_data.)
```

7.4. Source code organization

The LIP source tree has the following organization. Directories are in **boldface** type.

lip
- CMakeLists.txt
- INSTALL
- INSTALL.CMake
- LEGAL_NOTICE
- LIP_Ftype.h
- Makefile.in
- README
- RELEASE_NOTICE
- TESTING
- **aux_source**
- **build_cmake**
- **config**
- configure
- **data**
- **docs**
- **parallel_tests**
- release_checklist.txt
- **source**
- **test**
- **utility**

The contents of these files/directories are as follows:

`CMakeLists.txt` : top-level input file to `CMake` that controls the building of LIP. Read `INSTALL.CMake`.

`INSTALL` : instructions for building `liblip.a` and the test and utility codes.

`INSTALL.CMake` : instructions for building `liblip.a` and the test and utility codes using `CMake`.

`LEGAL_NOTICE` : standard legal notice.

`LIP_Ftype.h` : header file for defining Fortran compatible data types. This is here, rather than in **`source`**, because it is used by the `configure` script as a check for being in the top-level source code directory.

`Makefile.in` : This is a template for the top level `Makefile` for the package, that will be generated when `configure` is run. Read `INSTALL` first.

`README` : contains information on the contents of this top-level directory.

`RELEASE_NOTICE` : notice of the current version of LIP and how it differs from the previous version(s).

`TESTING` : instructions for running the test codes and interpreting the results.

`aux_source` : contains auxiliary functions and codes used by one or more of the LIP test or utility codes.

`build_cmake` : directory that can be used for building with `CMake`, since an out-of-source build is required for `CMake`; can be used for out-of-source build with `configure`, as well.

`config` : contains all scripts and files for configuring LIP with `Autoconf` tools.

`configure` : script for configuring the `Autoconf` build process. Read `INSTALL`.

`data` : contains various data sets used for testing parts of LIP.

`docs` : contains LIP documentation files, as well as tools for creating `user_docs` files from LIP source files.

`parallel_tests` : contains tests which exercise LIP using shared memory.

`release_checklist.txt`: a checklist used by the maintainers of LIP when a new version is about to be released.

`source` : contains the actual source code for LIP. The contents of this directory will be discussed in the following sections.

`test` : contains various programs for testing parts of LIP. The individual test codes are in separate subdirectories, each with its own `README` file.

`utility` : contains various LIP utility programs and procedures for building and testing them. The individual utility codes are in separate subdirectories, each with its own `README` file.

7.5. LIP data structures and setup functions

The following files deal with defining, initializing, populating, and interrogating LIP interpolation objects. Refer to `lip/docs/user_docs_setup` for the user interfaces for the setup functions and `lip/docs/user_docs_getters` for the user interfaces for the interrogation functions.

`lip_setup.c` : functions to initialize, populate, and free a LIP interpolation object.

`LIP_setup.h` : definition of the `LIP_interp` data type, which sets up a LIP interpolation object, and prototypes for the LIP setup functions.

`lip_setup_interp.c` : comprehensive function to initialize and populate a LIP interpolation object. This uses several functions from `lip_setup.c`. It is in a separate file and its prototype is in `LIP_proto.h`, because it calls functions from several other source files whose prototypes are declared there. (See Section 7.3, above, for more details on this function.)

`lip_utility.c` : functions to interrogate the contents of a LIP interpolation object.

`LIP_utility.h` : prototypes for the functions in `lip_utility.c`. (Of course, this `#include`'s `LIP_setup.h`.)

Routines included in file `lip_setup.c` are:

`lip_init_interp()` : initialize an interpolation object.

`lip_valid_setup()` : validate the object's `setup-type`.

`lip_add_data()` : add data to an interpolation object.

`lip_add_1der()` : add one derivative (for 1-D data) to an interpolation object.

`lip_add_2der()` : add two derivatives (for 2-D data) to an interpolation object.

`lip_add_twists()` : add twists (for 2-D data) to an interpolation object.

`lip_get_nbasfcns()` : get value of `nbasfcns` (called `ncoef` in the table in Section 1.3) for provided `int_type`. (N.B.: this function is also used to check for a valid `int_type` value.)

`lip_add_coeff()` : add full coefficient array (calculated elsewhere) to an interpolation object.

`lip_free_interp()` : free space for all array fields, then de-allocate the object.

Routines included in file `lip_utility.c` are:

The following functions return specified fields from a `LIP_interp` object:

`lip_get_setup_type()` : get the `setup_type` value from an interpolation object. (See Appendix A for allowable values.)

`lip_get_xname()` : get the `xname` (name associated with the first independent variable) from an interpolation object.

`lip_get_nx()` : get the `nx`-value from an interpolation object.

`lip_get_x()` : get the `x` array from an interpolation object.

`lip_get_order_x()` : get the `order_x` value from an interpolation object.

`lip_get_yname()` : get the `yname` (name associated with the second independent variable) from an interpolation object.

`lip_get_ny()` : get the `ny`-value from an interpolation object.

`lip_get_y()` : get the `y` array from an interpolation object.

`lip_get_order_y()` : get the `order_y` value from an interpolation object.

`lip_get_fname()` : get the `fname` (name associated with the dependent variable) from an interpolation object.

`lip_get_fval()` :get the `fval` (function value) array from an interpolation object.

`lip_get_dfdx()` : get the `dfdx` array from an interpolation object.

`lip_get_dfdy()` : get the `dfdy` array from an interpolation object.

`lip_get_twists()` : get the `twists` array from an interpolation object.

`lip_get_coeff()` : get the `coeff` array from an interpolation object.

`lip_get_int_type()` : get the `int_type` value from an interpolation object. (See Appendix A for allowable values.)

Caution: In the case of an array field, the return value from the associated function is a pointer, not a copy of the array.

Other utility functions:

`lip_set_int_type()` : set the `int_type` value in an existing interpolation object.

`lip_print_interp()` : print the contents of an interpolation object in a readable format.

7.6. LIP general utility functions

The following files contain various general utilities for LIP.

`lip_int_util.c` : miscellaneous utility functions used by one or more of the LIP evaluators or coefficient generators.

`LIP_int_util.h` : prototypes for the LIP interpolation utilities contained in file `lip_int_util.c`.

`lip_vers_date.c` : functions to return the version number and date for the current LIP package. (This `#include`'s `LIP_config.h`, which is created by the LIP build process.) Prototypes for these functions are in `LIP_int_util.h`.

`lip_datarev.c` - function to reverse one or both independent variables of a 2-D data set. (This is a separate file so that it can be used both by LIP interpolation functions and utility `datarev`.) Its prototypes are in `LIP_int_util.h`.

`lip_error.c` : defines the LIP error handling functions.

`LIP_error.h` : prototypes for the functions in `lip_error.c`. (This is `#included` in `LIP_setup.h` and `LIP_int_util.h`.)

Routines included in file `lip_int_util.c` are:

`lip_dcopy()` : copy the contents of a `Real8` array to another.

`lip_dswap()` : swap the contents of two `Real8` arrays.

`lip_fun3c()` : three-point difference derivative approximation.

`lip_p3d()` : compute the derivative of the cubic polynomial that interpolates a given set of data points.

`lip_cubic_derivs()` : compute derivative estimates suitable for piecewise cubic interpolation in a given (x,y) data table. (See Section 3.2.1.)

`lip_cubic_der2_cell()` : a modification of `lip_cubic_derivs` that computes only two derivative estimates and requires only 6 input f-values.

`lip_cubic_est()` : compute “standard” derivative estimate for `lip_cubic_derivs`.

`lip_cubic_est2()` : a modification of `lip_cubic_est` that only requires five f-values, not a whole slice.

`lip_twist_est()` : estimate the “twist” at a point from independent variable arrays and user-supplied first derivative arrays.

`lip_twist_est2()` : a modified version of `lip_twist_est` that returns estimates for all four box corners.

`lip_twist_est_box()` : a box-oriented version of `lip_twist_est` that uses only local derivative values.

`lip_deriv_est_box()` : compute derivative estimates needed by `lip_twist_est_box`.

`lip_hbasisf()` : evaluate the four Hermite basis functions at a point in [0,1].

`lip_hbasisd()` : evaluate derivatives of the four Hermite basis functions at a point in [0,1].

`lip_mach()` : emulate SLATEC floating point properties function.

`lip_fsign()` : emulate Fortran's SIGN function.

`lip_sign_test()` : emulate the PCHIP sign-testing function.

`lip_lookup()` : LIP table look-up routine. (See Section 1.5.)

Routines included in file `lip_vers_date.c` are:

`lip_package_version()` : return the interpolation package version number.

`lip_package_date()` : return the interpolation package date.

Routines included in file `lip_datarev.c` are:

`lip_datarev()` : reverses the provided data set in either or both independent variables.

`lip_rev1()` : reverses the order of a 1-D array. (Used by the above.)

Routines included in file `lip_error.c` are:

`lip_error_print()` : print an error message to the LIP global error message string `lip_errmsg`.

7.7. LIP coefficient generation

The following files contain functions for computing interpolation coefficients for a LIP interpolation object. Refer to `lip/docs/user_docs_interp` for the user interfaces for the user-callable coefficient generation and interpolation functions. See Appendix A for a list of supported interpolation types.

`LIP_proto.h` : prototypes for the LIP coefficient generation and interpolation functions. (This contains `#include`'s for `LIP_setup.h`, `LIP_utility.h`, and `LIP_int_util.h`, so that a user code need only include `LIP_proto.h`.)

`lip_coeff.c` : functions to calculate interpolation coefficients for a LIP interpolation object.

`lip_setup_bimond.c` : separate code for "bimond", the `LIP_MONO` coefficient setup (two-dimensional case).

`pbhpm.c` : functions that do the actual work for `lip_setup_bimond`. The primary function is PBHpm (Piecewise Bicubic Hermite interpolation that preserves monotonicity).

`LIP_PBH.h` : prototypes for the PBH functions in `pbhpm.c`.

`pchsubs.c` : two functions from the old PCHIC package used by the PBH functions.

7. Software Organization

`LIP_PCH.h` : prototypes for the functions in `pchsubs.c`.

Routines included in file `lip_coeff.c` are:

- `lip_set_critvals()` : set critical values for `LIP_CUBM`. (See Section 3.3.)
- `lip_get_critvals()` : retrieve critical values for `LIP_CUBM`. (Used by `lip_calc_coeff`.)
- `lip_calc_coeff()` : compute a coefficient array and add it to an existing LIP interpolation object. This is the primary user-callable function in this file. (If `int_type= LIP_CUBM`, it is necessary to call `lip_set_critvals` first.)
- `lip_setup_linear()` : calculate linear interpolation coefficients.
- `lip_setup_cubic()` : calculate cubic interpolation coefficients.
- `lip_setup_hermit()` : calculate cubic Hermite interpolation coefficients.
- `lip_setup_monder()` : calculate piecewise monotonic cubic Hermite interpolation coefficients.
- `lip_monomod1D()` : used by `lip_setup_monder` to modify derivatives to be suitable for piecewise monotonicity.
- `lip_1D_end_mods()` : post-process previously setup 1-D cubic to enforce user-specified boundary derivatives.
- `lip_setup_bilinear()` : calculate bilinear interpolation coefficients (`LIP_LIN`).
- `lip_setup_bicubic()` : calculate interpolation coefficients for the 12-term bicubic form (`LIP_CUB`).
- `lip_bicubic_matrix()` : set up and factor the `lip_setup_bicubic` system matrix.
- `lip_bicubic_rhs()` : compute the right-hand-side vector for the `lip_setup_bicubic` linear system.
- `lip_factor()` : compute the LU factorization of a matrix. (Used by `lip_bicubic_matrix`.)
- `lip_solve()` : solve a linear system, given its LU factorization. (Used by `lip_setup_bicubic`.)
- `lip_setup_bicubic2p()` : calculate bicubic interpolation coefficients for two-phase data (`LIP_CUBM`). (A modification of `lip_setup_bicubic`.)

`lip_flat_for_2p()` : function used by `lip_setup_bicubic2p` to detect two-phase data.

`lip_setup_biherm()` : calculate bicubic Hermite interpolation coefficients (`LIP_HERM`).

`lip_bicubic_derivs()` : calculate derivative estimates suitable for bicubic Hermite interpolation (`int_type=LIP_HERM` or `LIP_MONO`)

`lip_coeff_biherm()` : perform the actual coefficient calculations after `lip_setup_biherm` estimates first derivatives. (Also used by `lip_setup_bimond`.)

`lip_coeff_bh_one()` : set up the bicubic Hermite interpolation coefficients for one mesh box. (Used by `lip_coeff_biherm`.)

`lip_setup_biquad()` : calculate biquadratic interpolation coefficients (`LIP_QUAD`).

Routines included in file `lip_setup_bimond.c` are:

`lip_setup_bimond()` : calculate monotone bicubic Hermite interpolation coefficients (`LIP_MONO`). (This is basically an interface routine for `PBHpm`, described below.)

File `pbhpm.c` contains the C version of the Fortran function `PBHPM`, along with all of its subsidiary routines. Routines included are:

`PBHpm` : main control routine for BIMOND5 coefficient setup.

The names for its subsidiary routines were inherited from the Fortran version.

`pbhinit_` : PBHCOM Initialization Routine.

`pbhpset_` : Re-set `IPRINT`, primarily for debugging. (Added at version 1.6. Not part of Fortran version.)

`pbhm1a_` : Step 1 of bicubic Hermite derivative algorithm.

`pbhcz_` : BIMOND Compress Zero string routine.

`pbhm2b_` : Step 2 of bicubic Hermite derivative algorithm.

`pbhm3a_` : Step 3 of bicubic Hermite derivative algorithm.

`pbhxmb_` : Piecewise Bicubic Hermite X-Monotonicity checker.

`pbhymb_` : Piecewise Bicubic Hermite Y-Monotonicity checker.

`pbhsda_` : BIMOND String Decomposition Routine.

`pbhsg_` : Piecewise Bicubic Hermite SiGn routine.

`pbhtpa_` : BIMOND Transition element Processor.

`pbhts_` : Modified BIMOND Two-sweep Algorithm.

`pbhm4a_` : Step 4 of bicubic Hermite derivative algorithm.

`pbhtw_` : Piecewise Bicubic Hermite TWist routine.

`adc3p_` : Approximate Derivative by Centered 3-Point formula.

File `pchsubs.c` contains the C versions of needed functions from the Fortran package `PCHIP`. Routines included are:

`pchcs8_` : `PCHIC8` Monotonicity Switch Derivative Setter. (Used by `PBHpm` to adjust initially estimated derivatives.)

`pchsw8_` : `PCHCS8` Switch Excursion Limiter. (Used by `pchcs8_`.)

7.8. LIP interpolation functions

The following files contain functions for forward and inverse interpolation using a LIP interpolation object. Refer to `lip/docs/user_docs_interp` for the user interfaces for the user-callable coefficient generation and interpolation functions. See Appendix A for a list of supported interpolation types.

`LIP_proto.h` : prototypes for the LIP coefficient generation and interpolation functions. (This contains `#include`'s for `LIP_setup.h`, `LIP_utility.h`, and `LIP_int_util.h`, so that a user code need only include `LIP_proto.h`.)

`lip_eval1D.c` : LIP one-dimensional evaluation functions.

`lip_RatInterp.c` : separate code for (univariate) rational interpolation.

`lip_eval2D.c` : LIP two-dimensional evaluation functions.

`lip_BiRatInterp.c` : separate code for birational interpolation.

`lip_inverse.c` : functions for inverting two-dimensional interpolants. (Currently the package only supports inversion in the second independent variable.)

Routines included in file `lip_eval1D.c` are:

`lip_evalu_univar()` : general univariate interpolant evaluator. (This is the user-callable function for 1-D interpolation.)

`lip_evalu_cell()` : univariate interpolant evaluator with known cell indices.

The following four functions are called by `lip_evalu_cell`:

`lip_evalu_linear()` : used when `int_type=LIP_LIN`.

`lip_evalu_cubic()` : used when `int_type=LIP_CUB` or `LIP_CUBM`.

`lip_evalu_hermit()` : used when `int_type=LIP_HERM` or `LIP_MONO`.

`lip_evalu_rat()` : used when `int_type=LIP_RAT`.

(Note that `lip_evalu_rat` and its evaluator `lip_RatInt` are in separate file `lip_RatInterp.c`.)

The following auxiliary function is also in this file:

`lip_cells()` : process an index array for `lip_evalu_univar`.

Routines included in file `lip_eval2D.c` are:

`lip_set_bimond_exp()` : set the mesh expansion parameters for `lip_evalu_bimond`. (See end of Section 4.3.)

`lip_get_bimond_exp()` : get the mesh expansion parameters to be used by `lip_evalu_bimond`.

`lip_evalu_bivar()` : general bivariate interpolant evaluator. (This is the user-callable function for 2-D interpolation.)

`lip_evalu_box()` : bivariate interpolant evaluator with known box indices.

The following seven functions are called by `lip_evalu_box`:

`lip_evalu_bilinear()` : used when `int_type=LIP_LIN`.

`lip_evalu_bicubic()` : used when `int_type=LIP_CUB`.

`lip_evalu_bicubic2p()` : used when `int_type=LIP_CUBM`.

`lip_evalu_biherm()` : used when `int_type=LIP_HERM`.

`lip_evalu_bimond()` : used when `int_type=LIP_MONO`.

`lip_evalu_biquad()` : used when `int_type=LIP_QUAD`.

`lip_evalu_birat()` : used when `int_type=LIP_RAT`.

(Note that `lip_evalu_birat` and its evaluator `lip_BiRatInt` are in separate file `lip_BiRatInterp.c`.)

The following auxiliary functions are also in this file:

`lip_boxes()` : process a pair of index arrays for `lip_evalu_bivar`.

`lip_coeff_bh_one2()` : a modified version of `lip_coeff_bh_one` for use only by evaluators.

`lip_evalu_bihermite()` : does the evaluations for `lip_evalu_biherm` and `lip_evalu_bimond`.

Routines included in file `lip_inverse.c` are:

`lip_inverse_vals()` : general inverter. (This is the user-callable function for inverse interpolation.)

`lip_inv_coeff()` : used when full coefficient setup has been done.

`lip_inv_partial()` : used for partial coefficient setup case.

`lip_inv_general()` : general `int_type`-independent inverter.

The following functions are called by `lip_inv_coeff`:

`lip_inv_bilinear()` : used when `int_type`= `LIP_LIN`.

`lip_inv_biquad()` : used when `int_type`= `LIP_QUAD`.

`lip_inv_bicubic()` : used when `int_type`= `LIP_CUB` or `LIP_CUBM`.

`lip_inv_biherm()` : used when `int_type`= `LIP_HERM` or `LIP_MONO`.

The following functions are called by `lip_inv_partial`:

`lip_inv_bilin()` : used when `int_type`= `LIP_LIN`.

`lip_inv_bicub()` : used when `int_type`= `LIP_CUB`.

`lip_inv_biher()` : used when `int_type`= `LIP_HERM`.

`lip_inv_bimon()` : used when `int_type`= `LIP_MONO.`

The following auxiliary functions are also in this file:

`lip_inv_comp()` : checks for an allowable pair of `int_type` values for `lip_inv_coeff`.

`lip_in_intrv()` : used by `lip_inv_biquad` to check whether a point is in a specified interval.

`lip_coef_box_bicub()` : used by `lip_inv_bicub` to compute local interpolation coefficients as needed.

`lip_coef_box_biher()` : used by `lip_inv_biher` to compute local interpolation coefficients as needed.

`lip_coef_box_bimon()` : used by `lip_inv_bimon` to compute local interpolation coefficients as needed.

`lip_solve_bicub()` : used by both `lip_inv_bicubic` and `lip_inv_bicub` to solve the univariate cubic equation that determines a particular bicubic inverse value.

`lip_solve_biher ()` : used by `lip_inv_biherm`, `lip_inv_biher` and `lip_inv_bimon` to solve the univariate cubic equation that determines a particular bicubic Hermite inverse value.

`lip_coef_strip_bimon()` : used by `lip_inv_bimon` to compute a strip of interpolation coefficients in the data-only case.

`lip_inv_jlim()` : used by `lip_inv_bimon` when `npts=1.`

7.9. LIP programming conventions

The following programming conventions are used in most LIP source files.

1. Each function definition begins with a "`Start of function`" block and ends with an "`End of function`" block.
2. Each nontrivial function definition contains:
   a. a "`Declare local variables`" block and
   b. a "`Begin executable statements`" block.
3. The normal return value of a function is zero. A positive value indicates a warning (see individual function preamble), whereas a negative value indicates a fatal error. In this case, error messages are written by `lip_error_print` to global error message string `lip_errmsg`, which the calling program should declare and deal with in an appropriate manner.

   A complete list of the LIP fatal error numbers is given in file `lip/source/Error_nos`. (See Appendix G.) This includes the name of the function that generates each error.

8. Examples of Package Usage

8.1. Standard (bivariate) example

This section contains a complete program that illustrates the use of LIP to solve an interpolation problem. It first reads a set of sample data using function `readeos` (see Appendix C). It then sets up two LIP interpolation objects for this data, one using bicubic Hermite interpolation (`int_type=LIP_HERM`) and the other using bimond (`int_type=LIP_MONO`). It first evaluates these interpolants on the input data mesh and checks the interpolation accuracy and signs of derivatives at the mesh points. Then it evaluates each at a selection of points in the interior of a mesh box. For the provided data (see Appendix D), mesh box (4,0) is known to give interpolation procedures trouble, so several interior points in that box are selected to study the behavior of the two interpolants there. The test results are given in Appendix E.

Note: To improve the readability of the example code, the change record, optional debug printouts, and lines that test returned values from LIP functions have been omitted here. They are included in the available source code, which is in `lip/test/sample`.

```
/**************************************************
*                                                 *
*    Sample code illustrating the use of LIP      *
*                                                 *
**************************************************/

#include <math.h>
#include <float.h>
#include <stdlib.h>
#include <stdio.h>

/* The following is for the LIP test build procedure. */
#ifdef HAVE_CONFIG_H
#include "LIP_config.h"
#endif

#include "LIP_macros.h"  /* For various macro definitions. */
#include "LIP_proto.h"   /* LIP function prototypes. */

/* Define maxabs function (min and max defined in LIP_macros.h). */
#define maxabs(A,B) ( max( fabs(A), fabs(B) ) )

/* Tolerance for relative error tests. */
#define ERRTOL 1.0e-14

char errmsg[2*MAXLINE];  /* Test global error message string (2 lines). */
char lip_errmsg[MAXLINE];  /* LIP global error message string. */

/* Prototype for data read function. */
Integer readeos(const char *filename,
                Integer *nx, Real8 **x, Integer *ny, Real8 **y,
                Real8 **f, char *fname);
```

```
/**********************/
/* Start of main code */
/**********************/

int main()

/***********************************************************************
************************************************************************
*
* This sample code defines both a BIHERM and BIMOND interpolant to an
* input data set.  It first evaluates both on the input mesh and
* verifies that both reproduce the data within machine precision.
* It then shows that the BIHERM interpolant is not monotonic, but the
* BIMOND one is.
*
* Because EOS (equation of state) data is used for this example,
* variable names rho and t are used instead of x and y.
*
************************************************************************
************************************************************************
/

/* Implementation Note:                                                */
/*   Function values computed by BIHERM are denoted by p1val and the */
/*   associated derivatives by dp1dr, dp1dt.                          */
/*   Function values computed by BIMOND are denoted by p2val and the */
/*   associated derivatives by dp2dr, dp2dt.                          */

{

 /* Declare data variables. */
    char filename[80];  /* Name of input file. */
    Integer nrho, nt;
    Real8   *f, *rho, *t;
    char fname[8];

 /* Declare two LIP interpolation objects. */
    LIP_interp *interp_herm;  /* For the BIHERM interpolant. */
    LIP_interp *interp_mono;  /* For the BIMOND interpolant. */
    LIP_meth int_type;
    Integer job_flag, retfree;

 /* Declare interpolation variables. */
    Integer extr_flag=0;  /* Set for constant extrapolation. */
    Integer npts;
    Logical calcf, calcdr, calcdt;
    Real8   *rhoval, *tval;
    Real8   *dp1dr, *dp1dt, *p1val;
    Real8   *dp2dr, *dp2dt, *p2val;

 /* Declare other variables. */
    Integer i, j, k, retval;
    Integer nbad1, nbad2;
    Real8   drho, dt, error, rhosave;


/* Begin executable statements */
```

```
/* ========================== */

    printf ("\n\t LIP sample code");
    printf ("\n\t-----------------\n");

    /* Get name of input file and read data from it. */

    printf("\n Type file name.\n");
    if ( scanf("%s", filename) != 1) {
        printf("\n*** Trouble reading filename.\n\n");
        printf("     Aborting run.\n");
        return -999;
    }
    printf("\n");   /* Skip a line before any readeos output. */

    retval = readeos(filename, &nrho, &rho, &nt, &t, &f, fname);
    /* Note: on successful return, readeos will have allocated space */
    /*       for arrays rho, t, f.                                   */

    /* Should check value of retval and take appropriate action. */

    printf("\n readeos returned nrho =%5i, nt =%5i.\n", nrho, nt);

    /* Check that enough data has been read to define coefficients. */
    if ( nt<4 || nrho<4 ) {
        printf("\n ...Bad parameter value(s): nt = %i, nrho = %i\n",
               nt, nrho);
        retval = -1;
        goto Abort;
    }

    printf("\n\tResults for %s table from file %s\n", fname, filename);

/*-----------------------------------------------------------------*/
/* Initialize pointers here, in case of error in lip_setup_interp. */

    rhoval = NULL;
    tval = NULL;
    p1val = NULL;
    dp1dr = NULL;
    dp1dt = NULL;
    p2val = NULL;
    dp2dr = NULL;
    dp2dt = NULL;

/*---------------------------------------------------------*/
/* Allocate LIP interpolation objects, store data in them, */
/* and compute interpolation coefficients.                 */

    job_flag = 3;  /* Do full coefficient setup. */

    /* Note that the xname and yname values were chosen specifically */
    /* for the provided data set alplog, which has logged variables. */

    int_type = LIP_HERM;
    retval = lip_setup_interp( &interp_herm, "log(rho)", nrho, rho,
                               "log(T)", nt, t, fname, f, int_type,
                               job_flag );
```

```
    /* Should check value of retval and take appropriate action. */

    int_type = LIP_MONO;
    retval = lip_setup_interp( &interp_mono, "log(rho)", nrho, rho,
                               "log(T)", nt, t, fname, f, int_type,
                               job_flag );

    /* Should check value of retval and take appropriate action. */

/*-------------------------------------------*/
/* Set up variables for evaluating function */
/* and derivatives on data mesh.            */

    npts = nrho*nt;  /* Total number of data points. */
    rhoval = FMAKE_N( Real8, npts, "SAMPLE:rhoval" );
    tval = FMAKE_N( Real8, npts, "SAMPLE:tval" );
    p1val = FMAKE_N( Real8, npts, "SAMPLE:p1val" );
    dp1dr = FMAKE_N( Real8, npts, "SAMPLE:dp1dr" );
    dp1dt = FMAKE_N( Real8, npts, "SAMPLE:dp1dt" );
    p2val = FMAKE_N( Real8, npts, "SAMPLE:p2val" );
    dp2dr = FMAKE_N( Real8, npts, "SAMPLE:dp2dr" );
    dp2dt = FMAKE_N( Real8, npts, "SAMPLE:dp2dt" );
    if (  (rhoval==NULL) || (tval==NULL) ||
          (p1val==NULL) || (dp1dr==NULL) || (dp1dt==NULL) ||
          (p2val==NULL) || (dp2dr==NULL) || (dp2dt==NULL) ) {
        printf ("Trouble allocating storage for test arrays.\n");
        goto Done;
    }

    /* Pick up mesh in a 2-D array for evaluation. */
    k = 0;
    for (i=0; i<nrho; i++) {
        for (j=0; j<nt; j++) {
            rhoval[k] = rho[i];
            tval[k] = t[j];
            k++;
        }
    }
    if (k != npts) {
        printf ("Trouble setting up test mesh.");
        printf ("  Expect k = %i; got k = %i\n", npts, k);
        goto Done;
    }

/*-------------------------------------------------------------*/
/* Evaluate both interpolants and derivatives on the data mesh. */

    calcf=TRUE;  /* Calculate function values. */
    calcdr=TRUE; /* Calculate df/drho values. */
    calcdt=TRUE; /* Calculate df/dt values. */

    retval = lip_evalu_bivar(interp_herm, LIP_HERM, extr_flag,
                             rhoval, tval, npts, calcf, p1val,
                             calcdr, dp1dr, calcdt, dp1dt);
    /* Should check value of retval and take appropriate action. */

    retval = lip_evalu_bivar(interp_mono, LIP_MONO, extr_flag,
                             rhoval, tval, npts, calcf, p2val,
                             calcdr, dp2dr, calcdt, dp2dt);
```

```
    /* Should check value of retval and take appropriate action. */

/*-------------------------------------------------------------*/
/*  Loop over input mesh and compare computed results with data, */
/*  and also compare BIHERM with BIMOND at the data points.      */

    nbad1 = 0; /* Number of points where BIHERM values fail test. */
    nbad2 = 0; /* Number of points where BIMOND does not match. */
    k = 0;
    for (i=0; i<nrho; i++) {
        for (j=0; j<nt; j++) {

            error = fabs( f[j*nrho+i] - p1val[k] );
            if (error != 0.)  error /= maxabs( f[j*nrho+i], p1val[k] );
            if ( error > ERRTOL ) nbad1++;

          /* Compare with BIMOND interpolant. */
            error = fabs( p2val[k] - p1val[k] );
            if (error != 0.)  error /= maxabs( p2val[k], p1val[k] );
            if ( error > ERRTOL ) nbad2++;
            k++;

        }  /* End j-loop. */
    }  /* End i-loop. */

    /* Print summary of test results. */

    printf ("\n Tolerance for meshpoint accuracy tests = %e.\n", ERRTOL);
    printf (" %5i values of BIHERM interpolant failed to agree.\n",
            nbad1);
    printf (" %5i values of BIMOND interpolant failed to agree.\n",
             nbad2);

/*--------------------------------------------------------*/
/*  Loop over mesh and look for negative derivative values. */
/* (A monotone interpolant will have positive derivatives.) */

    nbad1 = 0; /* Number of negative p1 derivative values. */
    nbad2 = 0; /* Number of negative p2 derivative values. */
    k = 0;
    for (i=0; i<nrho; i++) {
        for (j=0; j<nt; j++) {

            if ( dp1dr[k] < 0. )  nbad1++;
            if ( dp1dt[k] < 0. )  nbad1++;

            if ( dp2dr[k] < 0. )  nbad2++;
            if ( dp2dt[k] < 0. )  nbad2++;

            k++;

        }  /* End j-loop. */
    }  /* End i-loop. */

    /* Print summary of derivative test results. */

   printf ("\n For monotonicity, all derivatives should be positive.\n");
   printf (" %5i values of BIHERM derivative at data points < 0.\n", nbad1);
```

```
   printf (" %5i values of BIMOND derivative at data points < 0.\n", nbad2);

/*----------------------------------------------------------*/
/*  The BIHERM interpolant is known to be nonmonotonic on   */
/*  mesh box (4,0).  Pick up several interior points which  */
/*  illustrate this point and evaluate both interpolants    */
/*  at those points.                                        */

    drho = rho[5] - rho[4];
    dt = t[1] - t[0];

    k = 0;
    for (i=1; i<4; i++) {
        rhosave = rho[4] + i*0.25*drho;
        for (j=1; j<4; j++) {
            rhoval[k] = rhosave;
            tval[k] = t[0] + j*0.25*dt;
            k++;
        }  /* End j-loop. */
    }  /* End i-loop. */

    npts = k;
    printf ("\n Evaluating at %i points, the quarter-points of mesh\n",
            npts);
    printf ("    box with lower-left corner at (rho,t)=(%.2f,%.2f).\n",
            rho[4], t[0]);

    /* Note that the calc flags are the same as before. */

    fflush (stdout);
    retval = lip_evalu_bivar(interp_herm, LIP_HERM, extr_flag,
                             rhoval, tval, npts, calcf, p1val,
                             calcdr, dp1dr, calcdt, dp1dt);
    /* Should check value of retval and take appropriate action. */

    fflush (stdout);
    retval = lip_evalu_bivar(interp_mono, LIP_MONO, extr_flag,
                             rhoval, tval, npts, calcf, p2val,
                             calcdr, dp2dr, calcdt, dp2dt);
    /* Should check value of retval and take appropriate action. */

    printf ("\n  rho     T        p1       dp1/drho   dp1/dT");
    printf ("      p2       dp2/drho   dp2/dT\n");
    for (k=0; k<npts; k++) {
        printf ("%6.4f%8.4f%11.3e%11.3e%10.3e%11.3e%11.3e%10.3e\n",
                rhoval[k], tval[k],
                p1val[k], dp1dr[k], dp1dt[k],
                p2val[k], dp2dr[k], dp2dt[k]);
    }

/*------------------*/

Done:
    printf("\n");
    fflush(stdout);

/* Free test variables */
```

```
    SFREE( rhoval );
    SFREE( tval );
    SFREE( p1val );
    SFREE( dp1dr );
    SFREE( dp1dt );
    SFREE( p2val );
    SFREE( dp2dr );
    SFREE( dp2dt );

/* Free the LIP_interp objects. */

    /* Free interp_herm. */
    retfree = lip_free_interp( &interp_herm );
    /* Should check value of retfree and take appropriate action. */

    /* Free interp_mono. */
    retfree = lip_free_interp( &interp_mono );
    /* Should check value of retfree and take appropriate action. */

/* Free space allocated by readeos. */

Abort:  /* Transfer point for errors during setup. */
    SFREE( rho );
    SFREE( t );
    SFREE( f );

    return retval;

}
/*******************/
/* End of main code */
/*******************/
```

8.2. A univariate example

The following code is presented for the benefit of users who have requested guidance in using LIP to interpolate one-dimensional data. It generates data from a cubic function. After setting up both linear and cubic interpolation objects, it evaluates each on a fine mesh and compares the computed results with the exact values. The output is in Appendix F.

Note: To improve the readability of the example code, the change record, optional debug printouts, and lines that test returned values from LIP functions have been omitted here. They are included in the available source code, which is in `lip/test/sample1D`.

```
/****************************************************************
*                                                               *
*     Sample code illustrating the use of LIP for 1-D data     *
*                                                               *
*****************************************************************
*
*  This code generates data from a cubic function.  It then defines
*  both a linear and cubic interpolant and compares results from each
*  to exact values on a refined mesh.
*
*----------------------------------------------------------------------
*/

#include <math.h>
#include <float.h>
#include <stdlib.h>
#include <stdio.h>

/* The following is for the LIP test build procedure. */
#ifdef HAVE_CONFIG_H
#include "LIP_config.h"
#endif

#include "LIP_macros.h"  /* For various macro definitions. */
#include "LIP_proto.h"   /* LIP function prototypes. */

char lip_errmsg[MAXLINE];  /* LIP global error message string. */

/* Parameters to define data and evaluation meshes. */
#define  NMAX     30    /* Maximum number of data points. */
#define  NPERMAX  10    /* Number of points per data interval. */
#define  NINTMX  (NMAX-1)*NPERMAX
#define  NEVLMX  NINTMX+1  /* Max number of evaluation points. */

/* Define maxabs function (max defined in LIP_macros). */
#define  maxabs(A,B)  (max(fabs(A),fabs(B)))

/* Tolerance for rel. vs abs. error. */
#ifdef SMALL
#undef SMALL
#endif
#define  SMALL  1.e-10
```

```
/*  Define cubic test function............. */
Real8  c0 = -0.9876;
Real8  c1 =  5.4321;
Real8  c2 = -1.234 ;
Real8  c3 =  0.567 ;
Real8  fcub (Real8 x) {
    return ( c0  +  x * (c1  +  x * (c2  +  x * c3)) ) ;
 }
Real8  dcub(Real8 x) {
    return ( c1  +  x * (2.*c2  +  x * (3.*c3)) ) ;
 }


/*  Define example code itself............. */

int main()
 {

 /* Declare data variables */
      Integer  n;
      Real8    x[NMAX], y[NMAX];

 /* Declare two LIP interpolation objects */
      LIP_interp  *interp_linear;  /* For the linear interpolant. */
      LIP_interp  *interp_cubic;   /* For the cubic interpolant. */
      LIP_meth int_type;
      Integer  job_flag, retval, zero=0;

 /* Declare interpolation variables */
      Integer  extr_flag=0;    /* Set for constant extrapolation. */
      Logical  calcfval=TRUE;     /* Do calculate function values. */
      Logical  calcdval=TRUE;     /* Do calculate derivative values. */
      Integer  ierr=0, neval, nint, nperint;
      Real8    dint[NEVLMX], dx, hint, xint[NEVLMX], yint[NEVLMX];

 /* Declare other variables */
      Integer  i, j;
      Integer  idmax, ifmax;
      Real8    denom, derr, derrmax, dtrue, ferr, ferrmax, ftrue;


/* Begin executable statements */
/* =========================== */

/*  Initialize.*/
      printf("\n\t LIP 1-D sample code\n");
      printf("\n\t---------------------\n");

      n = 20;             /* Number of data points. */
      nperint = NPERMAX;  /* Interpolation pts per data interval. */

/*  Set (arbitrary) data mesh.*/
      x[ 0] = -5. ;
      x[ 1] = -0.5;
      x[ 2] = -0.01;
      x[ 3] =  0.01;
      x[ 4] =  0.4;
```

```
      x[ 5] =  0.99;
      x[ 6] =  1.01;
      x[ 7] =  2.2;
      x[ 8] =  3.0;
      x[ 9] =  3.5;
      x[10] =  3.99999;
      x[11] =  4.0;
      x[12] =  4.8;
      x[13] =  5. ;
      x[14] =  6.5;
      x[15] =  6.9;
      x[16] =  7.1;
      x[17] =  8.2;
      x[18] =  9.5;
      x[19] = 10. ;

/*  Generate cubic data.*/
      printf ("\n Generating %i data points", n);
      printf (" over [%8.5f,%8.5f]\n", x[0], x[n-1]);
      for (i=0; i<n; i++) {
         y[i] = fcub(x[i]);
      }

/* -------------------------------------------------------- */
/*  Allocate LIP interpolation objects, store data in them, */
/*  and compute interpolation coefficients.                 */

/*  Note that the data is 1-D, with independent variable x, */
/*  and the dependent variable is y.  Here y is also used   */
/*  as a place-holder for the (unused) second independent   */
/*  variable in the lip_setup_interp calls.                 */

      job_flag = 3;  /* Do full coefficient setup. */

      int_type = LIP_LIN;
      retval = lip_setup_interp( &interp_linear, "x", n, x,
                                 "nil", zero, y, "fcub", y,
                                 int_type, job_flag );
     /* Should check value of retval and take appropriate action. */

      int_type = LIP_CUB;
      retval = lip_setup_interp( &interp_cubic, "x", n, x,
                                 "nil", zero, y, "fcub", y,
                                 int_type, job_flag );
     /* Should check value of retval and take appropriate action. */

/* ----------------------------------------------------------- */
/*  Compute interpolation mesh (nperint points per x-interval).*/

      printf (" Interpolating with densification nperint = %i.\n",
              nperint);
      nint = 0;
      for (i=0; i<n-1; i++) {
         hint = (x[i+1] - x[i])/nperint;
         for (j=0; j<nperint; j++) {
            dx = j*hint;
            xint[nint+j] = x[i] + dx;
          }
```

```
         nint = nint + nperint;
        }
      xint[nint] = x[n-1];
      neval = nint+1;

/* -------------------------------------------------------- */

/*  First test with linear interpolation coefficients.*/
      printf ("\n Testing linear case...\n");

/*  Evaluate the interpolant and its derivatives on fine mesh */
/*  using the linear interpolant.                             */
      int_type = LIP_LIN;
      ierr = lip_evalu_univar (interp_linear, int_type, extr_flag,
                               xint, neval, calcfval, yint,
                               calcdval, dint);
     /* Should check value of ierr and take appropriate action. */

/*  Assess the error in interpolated values. */
      ferrmax = -1;
      derrmax = -1;
      for (i=0; i<neval; i++) {
         ftrue = fcub(xint[i]);
         ferr = fabs(ftrue - yint[i]);
         denom = maxabs( ftrue, yint[i] );
         if (denom > SMALL)  ferr = ferr / denom;
         if (ferr > ferrmax) {
            ferrmax = ferr;
            ifmax = i;
      }
         dtrue = dcub(xint[i]);
         derr = fabs(dtrue - dint[i]);
         denom = maxabs( dtrue, dint[i] );
         if (denom > SMALL)  derr = derr / denom;
         if (derr > derrmax) {
            derrmax = derr;
            idmax = i;
      }
}
    printf("\n Largest error in linear interpolant =%15.7e at x = %f\n",
             ferrmax, xint[ifmax]);
    printf(" Largest error in linear derivative  =%15.7e at x = %f\n",
             derrmax, xint[idmax]);

/* ------------------------------------------------------------ */

/*  Repeat test with cubic interpolation coefficients.*/
      printf ("\n Testing cubic case...\n");

/*  Evaluate the interpolant and its derivatives on fine mesh */
/*  using the cubic interpolant.                              */
      int_type = LIP_CUB;
      ierr = lip_evalu_univar (interp_cubic, int_type, extr_flag,
                               xint, neval, calcfval, yint,
                               calcdval, dint);
     /* Should check value of ierr and take appropriate action. */
```

```
/*  Assess the error in interpolated values. */
      ferrmax = -1;
      derrmax = -1;
for (i=0; i<neval; i++) {
         ftrue = fcub(xint[i]);
         ferr = fabs(ftrue - yint[i]);
         denom = maxabs( ftrue, yint[i] );
         if (denom > SMALL)  ferr = ferr / denom;
         if (ferr > ferrmax) {
            ferrmax = ferr;
            ifmax = i;
      }
         dtrue = dcub(xint[i]);
         derr = fabs(dtrue - dint[i]);
         denom = maxabs( dtrue, dint[i] );
         if (denom > SMALL)  derr = derr / denom;
         if (derr > derrmax) {
            derrmax = derr;
            idmax = i;
      }
}
    printf("\n Largest error in cubic interpolant =%15.7e at x = %f\n",
             ferrmax, xint[ifmax]);
    printf(" Largest error in cubic derivative  =%15.7e at x = %f\n",
             derrmax, xint[idmax]);

/* ------------------------------------------------------------ */

 Done:

       /* Free the LIP_interp objects. */

      retval = lip_free_interp( &interp_linear );
    /* Should check value of retval and take appropriate action. */

      retval = lip_free_interp( &interp_cubic );
    /* Should check value of retval and take appropriate action. */

     exit(ierr);
 }
```

9. Possible Enhancements

9.1. Lookup improvements

Since EOS tabulation points are logarithmically spaced, it may be possible to further speed up the lookup phase by introducing a hash table or other device. It will be necessary to do this without incurring the expense of a logarithm or exponential call during evaluation. (Note: This may be out of place in a general interpolation library.)

9.2. Providing for user-supplied derivatives

Both the 12-term bicubic and the bicubic Hermite form use derivative values during coefficient setup. If such values are available, a facility should be provided to let them be supplied, rather than estimating them from the data.

9.3. Inversion in either independent variable

Historically, only inversion in the second independent variable (which is T for (ρ,T) EOS data) has been supported. It has been proposed that inversion in the first independent variable also be included in LIP, with appropriate safeguards for possible non-monotonic data.

References

[1] Fritsch, Frederick N., The LEOS Interpolation Package, Third Edition, UCRL-ID-148544-Rev.1 (March 12, 2003).

[2] Carlson, R.E., and F.N. Fritsch, "Monotone piecewise bicubic interpolation", SIAM J. Numer. Anal., Vol. 22, No. 2 (April 1985), pp.386–400.

[3] Carlson, R.E., and F.N. Fritsch, "An algorithm for monotone piecewise bicubic interpolation", SIAM J. Numer. Anal., Vol. 26, No. 1 (February 1989), pp.230–238. [The associated Fortran code, `BIMOND3`, was documented in UCID-21143 (August 1987).]

[4] Carlson, R.E., and F.N. Fritsch, "A bivariate interpolation algorithm for data which are monotone in one variable", SIAM J. Sci. Stat. Comput., Vol. 12, No. 4 (July 1991), pp.859–866. [The associated Fortran code, `BIMOND4`, was never formally documented.]

[5] Chase, Lila, EOS4 User Manual, Internal Report UCIR-1436a, Rev. 33 (July 7, 1999).

[6] Brown, S.A, and D. Braddy, PACT Users Guide, UCRL-MA-112087 (1993).

[7] Kerley, Gerald I., Rational Function Method of Interpolation, Informal Report LA-6903-MS, Los Alamos National Laboratory (August 1977).

Acknowledgements

I wish to thank David Young and Philip Sterne for supporting this project and for their patience with me while I tried to “get it right”. Special thanks are due to Ellen Hill for developing the configure/make procedure used to build a LIP library, and for providing quality assurance throughout this project. Thanks are also due to Larry Sanford for his help during the initial process of developing this stand-alone library, Rob Neely for suggestions on improving the C code that implements the library, Burl Hall, who provided the LANL birational interpolation code (see Section 6) and who developed the shared memory capability in LIP, as well as to several users for their suggestions.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344.

Appendix A. Values for LIP_interp Fields

This appendix lists the currently allowable values for certain fields in a `LIP_interp` object. These are defined in `LIP_setup.h`.

The `setup_type` (setup type) field may have the following values:

`LSU_UNINIT` : Uninitialized.

`LSU_INIT` : Initialized, but no data stored.

`LSU_DATA` : Only data provided.

`LSU_1DER` : Data + 1 derivative provided. [1-D data only]

`LSU_2DER` : Data + 2 derivatives provided. [2-D data only]

`LSU_3DER` : Data + 3 derivatives provided (both first partials plus twists). [2-D data only]

`LSU_COEF` : Full coefficient setup done.

These are defined in an `enum`, and have increasing values moving down the list.

The `int_type` (interpolation type) field may have the following values:

`LIP_INVALID` : Invalid, but initialized.

`LIP_LIN` : (Bi)linear interpolation. (See Section 2.)

`LIP_CUB` : "Standard" (bi)cubic interpolation. (See Section 3.)

`LIP_CUBM` : Modified bicubic interpolation (2-phase data). (See Section 3.3.)

`LIP_HERM` : (Bi)cubic Hermite interpolation. (See Section 4.)

`LIP_MONO` : Monotone (bi)cubic interpolation. (See Section 4.3.)

`LIP_QUAD` : Biquadratic interpolation. (See Section 5.)

`LIP_RAT` : (Bi)rational interpolation. (See Section 6.)

These are defined in an `enum`, and have increasing values moving down the list.

Appendix B. Adding a New Interpolation Method

The purpose of these notes is to discuss what is required for an interpolation method to be appropriate for LIP and to describe how to add such a new interpolator.

1. What kinds of interpolators can be supported?

The Livermore Interpolation Package (LIP) exists to support data provided on a two-dimensional rectangular mesh; namely, f(x,y): $x = x_0, x_1, ..., x_{nx-1}$; $y = y_0, y_1, ..., y_{ny-1}$. (Subscripting is from zero to be consistent with the C code.) Thus any interpolation method based on function (and possibly derivative) values on such a mesh is a potential candidate for a LIP interpolator. For full support, the following must be possible:

- The interpolation scheme can be factorized into a setup and an evaluation phase. The former computes a fixed number, nbasfcns, of interpolation coefficients for each mesh box. (Although we refer to *nbasfcns*, meaning *n*umber of *bas*is *f*un*c*tio*ns*, it is not required that the interpolant actually be a linear combination of basis functions.) At setup time these are stored in a coefficient array of size (nx–1)*(ny–1)*nbasfcns, where the original data mesh is nx by ny.

  Alternatively if only partial setup (see Section 1.4) is done, the evaluation phase must be able to compute the coefficient array for the current mesh box (or values of the interpolant) from the available data.

- A one-dimensional analog of the method, which is compatible with the 2-D method in the sense of Section 1.2, must exist. *Caution*: For this to work best, the "bixxx" functional form needs to be symmetric, so that the same univariate form applies to interpolation in either variable. That is, $f(x,y_{const}) = g(x)$ and $f(x_{const},y) = g(y)$, with the same univariate functional form g(x) – with different coefficients, of course.
- It must be possible to provide for inverse interpolation, as discussed in Section 1.8.

As a concrete example of a symmetric functional form, consider

$$\begin{aligned} f(x,y) = &\ (c_{00} + c_{10}\, x + c_{20} \ln(x) + c_{30}\, 1/x) \\ &+ (c_{01} + c_{11}\, x + c_{21} \ln(x) + c_{31}\, 1/x)\, y \\ &+ (c_{02} + c_{12}\, x + c_{22} \ln(x) + c_{32}\, 1/x) \ln(y) \\ &+ (c_{03} + c_{13}\, x + c_{23} \ln(x) + c_{33}\, 1/x)\, 1/y \text{ , where } x>0, y>0. \end{aligned}$$

This has nbasfcns = 16, the same as `LIP_HERM`. Obviously, if we fix x, this reduces to

$$g(y) = a_1 + b_1\, y + c_1 \ln(y) + d_1\, 1/y \text{ , where}$$

$$\begin{aligned} a_1 &= c_{00} + c_{10}\, x + c_{20} \ln(x) + c_{30}\, 1/x \text{ ,} \\ b_1 &= c_{01} + c_{11}\, x + c_{21} \ln(x) + c_{31}\, 1/x \text{ ,} \\ c_1 &= c_{02} + c_{12}\, x + c_{22} \ln(x) + c_{32}\, 1/x \text{ ,} \\ d_1 &= c_{03} + c_{13}\, x + c_{23} \ln(x) + c_{33}\, 1/x \text{ .} \end{aligned}$$

Similarly, if we fix y, this reduces to

$$g(x) = a_2 + b_2\, x + c_2 \ln(x) + d_2\, 1/x \text{ , where}$$

$$a_2 = c_{00} + c_{01}\, y + c_{02} \ln(y) + c_{03}\, 1/y \ ,$$
$$b_2 = c_{10} + c_{11}\, y + c_{12} \ln(y) + c_{13}\, 1/y \ ,$$
$$c_2 = c_{20} + c_{21}\, y + c_{22} \ln(y) + c_{23}\, 1/y \ ,$$
$$d_2 = c_{30} + c_{31}\, y + c_{32} \ln(y) + c_{33}\, 1/y \ .$$

Thus f has the desired symmetry property. The univariate analog has nbasfcns = 4.

2. Adding a new interpolation method to LIP.

In this section we refer to the new method as "bixxx interpolation", and its 1-D analog as "xxx interpolation".

Use existing functions in the files mentioned below as a guide to the assumed calling sequences. It is taken for granted that calling sequence documentation will be updated as necessary to reflect the new method. (A skeleton for the LIP standard prologue format is in file `lip/aux_source/skeleton`.)

1. Choose an `int_type` name `LIP_XXX` to designate bixxx interpolation and add its definition to `LIP_setup.h`. Note that this must be different from all previously defined interpolation type designators. Be sure to update `lip_get_nbasfcns` in `lip_setup_c` to test for this new `int_type`. (This may also require a change to `nbasfcns.c` and/or its output `nbas_out` in `lip/test/liptest`.)
2. As you start to write functions to implement the new method, as discussed below, refer to file `lip/source/Error_nos` to select new fatal error numbers. These should be distinct from those already in use. Note that coefficient setup functions use numbers -2nnn, bivariate evaluators use -3nnn, univariate evaluators use -4nnn, and inversion functions use -7nnn.
3. Write a function `lip_evalu_bixxx` that evaluates a previously set-up bixxx interpolant at an array of points. Add this to file `lip_eval2D.c`. Add a suitable call to it to `lip_evalu_box` in this file.
4. Write a function `lip_evalu_xxx` which evaluates a previously set-up xxx interpolant at an array of points. Add this to file `lip_eval1D.c`. Add a suitable call to it to `lip_evalu_cell` in this file.
5. Write a function `lip_setup_bixxx` that sets up the full coefficient array for bixxx interpolation. Add this to file `lip_coeff.c`. Add a suitable call to it to `lip_calc_coeff` in this file. (This is not necessary if the method is only implemented for the "partial setup" mode.)
6. Also write a function `lip_setup_xxx` that sets up the coefficient array for (univariate) xxx interpolation. Add this to file `lip_coeff.c` as well, and add a suitable call to it to `lip_calc_coeff` in this file. (Again, may not be necessary.)
7. Write a function `lip_inv_bixxx` which implements inverse bixxx interpolation in full coefficient setup mode, if appropriate. Add this to file `lip_inverse.c`.

Add a suitable call to it to `lip_inv_coeff` in this file. If partial setup is supported, add a function with a similar name to support it, via a new call from `lip_inv_partial`. (If neither is done, inversion may still be possible by using `lip_inv_general`.)

8. When done with implementation, add the new error numbers to the list in file `lip/source/Error_nos`.
9. Oh, yes! *Don't forget to change the LIP version number and date* in file `configure.in` (refer to the `README` file in directory `lip/config`). (To date, no policy has been established for version numbering, but this is a significant enough change that at least the first digit after the decimal point should be incremented.)
10. Update the documentation files (in directory `lip/docs`). The tools there will be helpful here. Note that it will be necessary to update file `prologue_interp` to reflect the current status. It may also be necessary to update file `names_interp` if you wish to include the new setup functions in the document. (Of course, you will need to update the prologues of the source files to which you add the new functions mentioned above.)
11. Add a bixxx option to appropriate test codes in `lip/test`. This may require adding new input and/or output files and modifying the `Run_this` scripts to run additional cases. If appropriate, also add a new test specific to this new method.

Note that items 3–7 will require addition of new prototypes to `LIP_proto.h`.

Appendix C. Data Read Function for Example

Following is the listing of the source code for function `readeos`, used by the example program of Section 8.1 to read its input data. The assumed format of the file is described in the initial comments. (We have omitted initial `#include`'s that don't aid understanding of the code. The actual code is in `lip/aux_source`.)

```
#include "LIP_macros.h"  /* For FMAKE_N. */
#include "LIP_Ftype.h"   /* For LIP data type definitions. */

extern char errmsg[];   /* Link to test global error message string. */

/****************************/
/* Start of function readeos */
/****************************/
Integer readeos(const char *filename,
                Integer *nx, Real8 **x, Integer *ny, Real8 **y,
                Real8 **f, char *fname)

/*********************************************************************
**********************************************************************
*  Function to read a function from a LIP 3-column data file.
*  (See note below on assumed format of the input file.)
*
   (Omitted comments on how readeos differs from read_eos.)
*
*  This function allocates array storage internally, after reading nx
*  and ny.  It is the responsibility of the calling program to call
*  SFREE for x, y, f when finished using them.  Note that to get the
*  connection right, these arrays are treated as pointers to pointers
*  here, and the ADDRESSES of the external arrays need to be sent in
*  where those arrays appear in the call list.
*
* Input variables:
*     filename  Name of the file to be read.
*
* Output variables:
*     nx        Number of x-values.
*     x         The x-values.
*     ny        Number of y-values.
*     y         The y-values.
*     f         The f-values.
*     fname     Will be the name of the dependent variable as read from
*               the file.
*  If there was trouble reading the data, the return value will be
*  negative.  On successful return, the data are stored in f in the
*  order required by LIP; that is, f[j*nx+i] is the value at point
*  x(i), y(j).
*
* Return value:  The return value, retval, should be zero.
*   The possible fatal error returns are:
*     retval =   -1 : cannot open input file.
*     retval =   -2 : problem scanning (fname,ny,nx) line for data set.
*     retval =   -3 : nx and/or nx <= 0.
```

```
*     retval =  -10 : bad y-value detected (nonrectangular mesh?).
*     retval = -100 : Trouble allocating memory.
*     retval = -200 : EOF encountered while reading Data ID Block.
*     retval = -201 : EOF encountered before end of a data set.
*  All error indications will result in a message to the global error
*  message string errmsg, as well.
*
* Assumed data format:
*  The first line of each table contains fname and two integers, in the
*  order ny and nx.  The fname field is assumed to be at most eight
*  characters.  For historical reasons, the second independent variable
*  (temperature for EOS data) is read before the first (EOS density).
*  This line is read via sscanf with format "%s %i %i".
*  The data then follow as y, x, f triples, with all the x-values for
*  a given y together.  (Again, note the order!)
*
*  This version allows a data ID block to be given before the data itself.
*  The first line of the block must be "*Begin ID Block" and the last line
*  of the block must be "*End ID Block".  The case of all letters must be
*  exactly as indicated here.  Any lines of text the creator of the table
*  wishes to use to identify the data may appear between these two lines.
*
*  This version looks for a file named "read_verbose".  If it exists, the
*  x and y arrays are printed, as well as the first and last values of
*  the dependent variable.  The data ID block, if present, will also be
*  printed in this case.
*
* Change record:
* (yymmdd means 20yy/mm/dd)
*  080806  Initial implementation by Fred N. Fritsch, LEOS Development
*          Team, from existing read_eos.
*  080812  Corrected errors in initial conversion and omitted argument
*          fread. (FNF)
*  080812  Modified to print error messages to errmsg, not stdout. (FNF)
*
**********************************************************************
*********************************************************************/
 {

/* Declare local variables */

   FILE *infile;
   char line[MAXLINE], invar[]="(none)";
   Integer i, j, retval;
   Real8   fin, xin, yin;
   Logical idblock, verbose;


/* Begin executable statements */
/* ========================== */

/* Look for file "read_verbose" and set flag verbose accordingly. */

   verbose = FALSE;
   if ( fopen("read_verbose", "r") != NULL )  verbose = TRUE;

/* Open the file to be read. */
```

```
   infile = fopen(filename, "r");
   if (infile == NULL) {
      sprintf(errmsg,
              "File  %s  does  not  exist  or  is  not  readable.\n",
filename);
      return (-1);
   }

/* Initialize. */

   idblock = FALSE;

   *x = (Real8 *) NULL;
   *y = (Real8 *) NULL;
   *f = (Real8 *) NULL;

Continue10:
      if (fgets(line, MAXLINE, infile) == NULL) {
         if (idblock) goto Read_error;
         else         goto Normal_exit;
      }

   /* Process optional data ID block. */

      if (strncmp(line, "*Begin ID Block", 15) == 0)  idblock = TRUE;
      if (idblock) {
         if (verbose)  printf(" %s", line);
         if (strncmp(line, "*End ID Block", 13) == 0) {
            idblock = FALSE;
            if (verbose)  printf(" \n");
         }
         goto Continue10;
      }

   /* Read initial data line (fname,ny,nx). */

      if (sscanf(line, "%s %i %i", fname, ny, nx) == EOF) {
         sprintf(errmsg,
                 " ERROR: Illegal initial data line:  %s\n", line);
         return (-2);
      } else {
         printf(" Reading variable %s", fname);
         printf(":   ny =%5i", *ny);
         printf(", nx =%5i\n", *nx);
         fflush(stdout);
      }
      if ( (*ny <= 0) || (*nx <= 0) ) {
         sprintf(errmsg,
                 "   ERROR: Bad ny=%i or nx=%i\n", *ny, *nx);
         sprintf(errmsg, "\t***  Aborting EOS read.\n");
         return (-3);
      }

   /* Allocate space for mesh variables. */

      *x = FMAKE_N(Real8, (*nx), "Readeos:x");
      *y = FMAKE_N(Real8, (*ny), "Readeos:y");
      if (*x == NULL || *y == NULL) goto Allocate_error;
```

```
/* Allocate space for function being read. */

   *f = FMAKE_N(Real8, (*nx)*(*ny), "Readeos:f");
   if (*f == NULL) goto Allocate_error;

/* Loop over y-values. */

   for (j = 0; j < *ny; j++) {

  /* Loop over x-values. */

      for (i = 0; i < *nx; i++) {
         if (fscanf(infile,"%le", &yin) == EOF) {
            strcpy(invar, "yin");
            goto Read_error;
         }
         if (fscanf(infile,"%le", &xin) == EOF) {
            strcpy(invar, "xin");
            goto Read_error;
         }
         if (fscanf(infile,"%le", &fin) == EOF) {
            strcpy(invar, "fin");
            goto Read_error;
         }
         if (i > 0) {
            if (yin != (*y)[j]) {
               sprintf(errmsg,
                       " Bad y-value.  Read %e; expected %e\n",
                       yin, (*y)[j]);
               retval = -10;
               goto Clean_up;
            }
         }
       /* Don't store y if already stored. */
         if (i == 0) {
            (*y)[j] = yin;
         }
       /* Store x. */
         (*x)[i] = xin;
       /* Store f. */
         (*f)[j*(*nx)+i] = fin;
      }
   }

   printf(" Finished reading %s data.\n", fname);
   if (verbose) {
      printf(" x-values:\n    ");
      for (i = 0;  i < *nx; i++) {
         printf("%15.7E", (*x)[i]);
         if (i == *nx-1) printf("\n");
         else if (i%5 == 4) printf("\n    ");
      }
      printf(" y-values:\n    ");
      for (j = 0;  j < *ny; j++) {
         printf("%15.7E", (*y)[j]);
         if (j == *ny-1) printf("\n");
         else if (j%5 == 4) printf("\n    ");
```

```
         }
         printf(" First and last %s-values:\n", fname);
         printf("    %15.7E%15.7E\n", (*f)[0], (*f)[(*ny-1)*(*nx)+*nx-1]);
      }
      printf("\n");

      /* Get rid of dangling end-of-line. */
      if (fgets(line, MAXLINE, infile) == NULL) goto Read_error;

/*  Normal exit. */

Normal_exit:
   printf(" Done reading data from file %s\n", filename);
   retval = 0;
   goto Clean_up;

/*  Error exits. */

Allocate_error:
   sprintf(errmsg,
           " ERROR: Cannot allocate space for one or more arrays.\n");
   retval = -100;
   goto Clean_up;

Read_error:
   if (idblock) {
      sprintf(errmsg, " ERROR: EOF while reading data ID block.\n");
      sprintf(errmsg, "       Missing '*End ID Block' line?\n");
      retval = -200;
   } else {
      sprintf(errmsg,
              " ERROR: EOF before end of data while reading %s for %s\n",
              invar, fname);
      retval = -201;
   }

Clean_up:
   /* Note: cannot free arrays here, because they are to be accessed */
   /*       and used by calling program.                             */

   /* The following error message should never appear. */
   if ( (*x == NULL) || (*y == NULL) || (*f == NULL) ) {
      printf(" ERROR: One or more NULL array pointers in readeos.\n");
   }

   return(retval);
}
/****************************/
/* End of function readeos */
/****************************/
```

Appendix D. Input File for Example

Following is the data file `alplog` read by the example program in Section 8.1. These data are increasing in both variables, although clearly not strictly increasing, for there are sections where the values are independent of ρ (second column). Note that the name of the file stands for "aluminum pressure with all variables logged". The "truncated" in the initial line of the ID Block means (1) this is but a small segment of a much larger EOS table and (2) the values have been truncated to two digits after the decimal point.

```
*Begin ID Block
       Truncated section of aluminum EOS table
This is the sample data set distributed with BIMOND3, converted
to the format expected by readeos. (FNF 5/24/2001)
Note that the data columns are log(T), log(rho), log(P).
*End ID Block
log(P)   6    10
     -2.30                -0.07                 -34.54
     -2.30                 0.33                 -34.54
     -2.30                 0.55                 -34.54
     -2.30                 0.69                 -34.54
     -2.30                 0.84                 -34.54
     -2.30                 0.93                 -34.54
     -2.30                 0.98                  -3.06
     -2.30                 1.02                  -2.86
     -2.30                 1.08                  -2.37
     -2.30                 1.13                  -1.89
     -1.61                -0.07                 -13.82
     -1.61                 0.33                 -13.82
     -1.61                 0.55                 -13.82
     -1.61                 0.69                 -13.82
     -1.61                 0.84                 -13.82
     -1.61                 0.93                  -2.68
     -1.61                 0.98                  -2.28
     -1.61                 1.02                  -1.92
     -1.61                 1.08                  -1.60
     -1.61                 1.13                  -1.30
     -0.92                -0.07                 -10.10
     -0.92                 0.33                 -10.10
     -0.92                 0.55                 -10.10
     -0.92                 0.69                 -10.10
     -0.92                 0.84                  -2.52
     -0.92                 0.93                  -1.88
     -0.92                 0.98                  -1.63
     -0.92                 1.02                  -1.39
     -0.92                 1.08                  -1.17
     -0.92                 1.13                  -0.95
     -0.51                -0.07                  -7.26
     -0.51                 0.33                  -7.26
     -0.51                 0.55                  -7.26
     -0.51                 0.69                  -4.82
     -0.51                 0.84                  -2.22
     -0.51                 0.93                  -1.56
```

```
-0.51          0.98          -1.32
-0.51          1.02          -1.10
-0.51          1.08          -0.90
-0.51          1.13          -0.71
-0.22         -0.07          -5.66
-0.22          0.33          -5.66
-0.22          0.55          -4.88
-0.22          0.69          -3.34
-0.22          0.84          -1.98
-0.22          0.93          -1.41
-0.22          0.98          -1.15
-0.22          1.02          -0.92
-0.22          1.08          -0.72
-0.22          1.13          -0.54
 0.           -0.07          -4.53
 0.            0.33          -4.13
 0.            0.55          -3.35
 0.            0.69          -2.73
 0.            0.84          -1.78
 0.            0.93          -1.28
 0.            0.98          -1.05
 0.            1.02          -0.81
 0.            1.08          -0.60
 0.            1.13          -0.41
```

Appendix E. Output for 2-D Example

Following is the result of running the example program in Section 8.1 with the input data given in Appendix D. Note three pieces of evidence that the BIHERM interpolant p_1 is not monotonic, even though the data are, but the BIMOND interpolant p_2 is monotonic. First, six negative derivative values were found when p_1 was evaluated on the data mesh, but none for p_2. Second, in the mesh box (4,0) chosen for special study we find one negative value for $\partial p_1/\partial \rho$, but none for p_2. Finally, we see that

$$p_1(0.8850,–2.1275) – p_1(0.8625,–2.1275) = (–29.67) – (–29.60) = –0.07,$$
$$p_1(0.9075,–2.1275) – p_1(0.8850,–2.1275) = (–27.52) – (–29.67) = +2.15,$$

so p_1 clearly is not monotonic in the first variable!

```
         LIP sample code
        -----------------


 Type file name.
alplog


 Reading variable log(P):   ny =    6, nx =   10
 Finished reading log(P) data.

 Done reading data from file alplog

 readeos returned nrho =   10, nt =    6.

        Results for log(P) table from file alplog

 Tolerance for meshpoint accuracy tests = 1.000000e-14.
     0 values of BIHERM interpolant failed to agree.
     0 values of BIMOND interpolant failed to agree.

 For monotonicity, all derivatives should be positive.
     6 values of BIHERM derivative at data points < 0.
     0 values of BIMOND derivative at data points < 0.

 Evaluating at 9 points, the quarter-points of mesh
    box with lower-left corner at (rho,t)=(0.84,-2.30).
```

Appendix E. Output for 2-D Example

rho	T	p1	dp1/drho	dp1/dT	p2	dp2/drho	dp2/dT
0.8625	-2.1275	-2.960e+01	-3.121e+01	4.602e+01	-2.721e+01	8.768e+01	4.156e+01
0.8625	-1.9550	-2.220e+01	6.974e+01	3.967e+01	-2.048e+01	1.347e+02	3.574e+01
0.8625	-1.7825	-1.596e+01	1.265e+02	3.251e+01	-1.516e+01	1.489e+02	2.510e+01
0.8850	-2.1275	-2.967e+01	3.579e+01	6.190e+01	-2.480e+01	1.171e+02	5.211e+01
0.8850	-1.9550	-2.012e+01	1.161e+02	4.885e+01	-1.677e+01	1.802e+02	4.043e+01
0.8850	-1.7825	-1.280e+01	1.499e+02	3.600e+01	-1.106e+01	1.993e+02	2.516e+01
0.9075	-2.1275	-2.752e+01	1.661e+02	7.083e+01	-2.238e+01	8.829e+01	6.273e+01
0.9075	-1.9550	-1.693e+01	1.682e+02	5.219e+01	-1.304e+01	1.365e+02	4.519e+01
0.9075	-1.7825	-9.406e+00	1.478e+02	3.537e+01	-6.930e+00	1.515e+02	2.523e+01

Appendix F. Output for 1-D Example

Following is the result of running the example program in Section 8.2. As one might expect, the linear interpolant is not very accurate, whereas the cubic interpolant is good nearly to machine precision.

```
     LIP 1-D sample code

    ---------------------

 Generating 20 data points over [-5.00000,10.00000]
 Interpolating with densification nperint = 10.

 Testing linear case...

 Largest error in linear interpolant =  5.7022637e-01 at x = -1.400000
 Largest error in linear derivative  =  6.6687884e-01 at x = -0.950000

 Testing cubic case...

 Largest error in cubic interpolant =  3.9106481e-13 at x = 4.240000
 Largest error in cubic derivative  =  6.0325540e-12 at x = 3.999994
```

Appendix G. LIP Fatal Error Numbers

The following is file `lip/source/Error_nos`.

```
        LIP Fatal Error Numbers

This file contains a list of the fatal error numbers, including the function
generating that error.  Note that most numbers are unique to a specific
function.  For the few that are not, the meaning is the same wherever it
originates.

 Number    Function            Meaning
 -1000   lip_init_interp     Input interp is NULL (un-allocated?).
 -1001   lip_init_interp     Trouble allocating space for one or more of the
                             xname, yname, or fname fields.
 -1100   lip_add_data        Input interp is NULL (un-allocated?).
 -1101   lip_add_data        Input interp->setup_type is LSU_UNINIT or has
                             an unrecognized value.
 -1102   lip_add_data        Both nx and ny are zero or either is negative.
 -1103   lip_add_data        Trouble resetting one or more of the xname,
                             yname, or fname fields.
 -1104   lip_add_data        One or more of the input array arguments is
                             NULL, despite input values of nx and ny.
 -1105   lip_add_data        Trouble allocating space for one or more of
                             the interp array fields.
 -1106   lip_add_data        One or both of the x, y arrays is not strictly
                             monotonic.
 -1200   lip_add_1der        Input interp is NULL (un-allocated?).
 -1201   lip_add_1der        Input interp->setup_type is not LSU_DATA.
 -1202   lip_add_1der        Input nx and/or ny do not agree with those
                             already set in interp.
 -1203   lip_add_1der        The nx and ny fields in interp are both positive,
                             so this is not 1-D data.
 -1204   lip_add_1der        One or more of the input array arguments is
                             NULL.
 -1205   lip_add_1der        Trouble allocating space for one or more of
                             the interp array fields.
 -1300   lip_add_2der        Input interp is NULL (un-allocated?).
 -1301   lip_add_2der        Input interp->setup_type is not LSU_DATA.
 -1302   lip_add_2der        Input nx and/or ny do not agree with those
                             already set in interp.
 -1303   lip_add_2der        The nx and/or ny field in interp is zero.
 -1304   lip_add_2der        One or more of the input array arguments is
                             NULL.
 -1305   lip_add_2der        Trouble allocating space for one or more of
                             the interp array fields.
 -1400   lip_add_twists      Input interp is NULL (un-allocated?).
 -1401   lip_add_twists      Input interp->setup_type is not LSU_2DER.
 -1402   lip_add_twists      Input nx and/or ny do not agree with those
                             already set in interp.
 -1403   lip_add_twists      The nx and/or ny field in interp is zero.
 -1404   lip_add_twists      Input array argument twists is NULL.
 -1405   lip_add_twists      Trouble allocating space for the interp twists
                             field.
 -1500   lip_add_coeff       Input interp is NULL (un-allocated?).
```

-1501	lip_add_coeff	Input interp->setup_type does not have one of the allowed values.
-1502	lip_add_coeff	Input nx and/or ny do not agree with those already set in interp.
-1503	lip_add_coeff	Input int_type is invalid for provided data.
-1504	lip_add_coeff	Input array argument coeff is NULL.
-1505	lip_add_coeff	Trouble allocating space for the interp coeff field.
-1701	lip_datarev	nx < 0 or ny < 0 (or both).
-1702	lip_datarev	rev_x or rev_y (or both) invalid.
-1800	lip_setup_interp	trouble creating interp.
-1801	lip_setup_interp	illegal value of job_flag.
-1802	lip_setup_interp	illegal or unsupported value of int_type.
-1803	lip_setup_interp	too few data points for linear interpolation.
-1804	lip_setup_interp	too few data points for rational interpolation.
-1805	lip_setup_interp	too few data points for cubic interpolation.
-1810	lip_setup_interp	trouble creating temporary workspace.
-1820	lip_setup_interp	error return from lip_monomod1D.
-1821	lip_setup_interp	error return from PBHpm.
-1999	lip_free_interp	Input interp is NULL (un-allocated?).
-2000	lip_calc_coeff	Input interp is NULL (un-allocated?).
-2001	lip_calc_coeff	Input interp setup_type does not have one of the allowed values.
-2002	lip_calc_coeff	Input nx and/or ny do not agree with those already set in interp.
-2003	lip_calc_coeff	Input nx and/or ny do not agree with those Input int_type is invalid for provided data.
-2005	lip_calc_coeff	Trouble allocating space for the coefficient array.
-2007	lip_calc_coeff	Trouble allocating extra lip_bicubic2p array (int_type=LIP_CUBM only).
-2008	lip_calc_coeff	Trouble getting rho_0, T_c for lip_bicubic2p (int_type=LIP_CUBM only).
-2009	lip_calc_coeff	Trouble allocating/storing modified lip_biquad arrays (int_type=LIP_QUAD only).
-2111	lip_setup_linear	n < 2.
-2121	lip_setup_cubic	n < 4.
-2124	lip_setup_cubic	trouble allocating temporary storage.
-2151	lip_setup_hermit	n < 4.
-2154	lip_setup_hermit	trouble allocating temporary storage.
-2171	lip_setup_monder	n < 4.
-2174	lip_setup_monder	trouble allocating temporary storage.
-2175	lip_setup_monder	trouble in pchcs8_ (should never happen).
-2174	lip_monomod1D	trouble allocating temporary storage.
-2175	lip_monomod1D	trouble in pchcs8_ (should never happen).
-2180	lip_1D_end_mods	invalid interp or setup_type is not LSU_COEF.
-2181	lip_1D_end_mods	invalid value of which.
-2182	lip_1D_end_mods	int_type in interp is not one of LIP_CUB, LIP_HERM, or LIP_MONO. (Not a cubic.)
-2185	lip_1D_end_mods	int_type=LIP_MONO, which>0, and one or both derivative values are incompatible with monotonicity.
-2186	lip_1D_end_mods	fatal error return from lip_cubic_der2_cell.
-2201	lip_setup_bilinear	nx < 2.
-2202	lip_setup_bilinear	ny < 2.
-2203	lip_setup_bilinear	both nx and ny were invalid.
-2301	lip_setup_bicubic	nx < 4.
-2302	lip_setup_bicubic	ny < 4.

-2303	lip_setup_bicubic	both nx and ny were invalid.
-2321	lip_setup_bicubic	trouble allocating space for linear system.
-2322	lip_setup_bicubic	coefficient setup failed due to a singular matrix (should never happen with this version).
-2323	lip_setup_bicubic	error return from lip_bicubic_rhs.
-2361	lip_bicubic_rhs	i does not satisfy 0 <= i < nx.
-2362	lip_bicubic_rhs	j does not satisfy 0 <= j < ny.
-2363	lip_bicubic_rhs	both i and j were invalid.
-2365	lip_bicubic_rhs	trouble allocating temporary workspace.
-2370-m	lip_bicubic_rhs	m>0 "Illegal value of icell" error returns were received from lip_cubic_der2_cell, which can only happen if ibox and/or jbox is bad.
-2401	lip_setup_bicubic2p	nx < 4.
-2402	lip_setup_bicubic2p	ny < 4.
-2403	lip_setup_bicubic2p	both nx and ny were invalid.
-2421	lip_setup_bicubic2p	trouble allocating space for linear system.
-2422	lip_setup_bicubic2p	coefficient setup failed due to a singular matrix (should never happen with this version).
-2423	lip_setup_bicubic2p	error return from lip_bicubic_rhs.
-2501	lip_setup_biherm	nx < 4
-2502	lip_setup_biherm	ny < 4.
-2503	lip_setup_biherm	both nx and ny were invalid.
-2521	lip_setup_biherm	trouble allocating temporary workspace.
-2501	lip_bicubic_derivs	nx < 4.
-2502	lip_bicubic_derivs	ny < 4.
-2503	lip_bicubic_derivs	both nx and ny were invalid.
-2525	lip_bicubic_derivs	trouble allocating temporary workspace.
-2531	lip_coeff_biherm	nx < 4.
-2532	lip_coeff_biherm	ny < 4.
-2533	lip_coeff_biherm	both nx and ny were invalid.
-2561	lip_coeff_bh_one	i does not satisfy 0 <= i < nx.
-2562	lip_coeff_bh_one	j does not satisfy 0 <= j < ny.
-2563	lip_coeff_bh_one	both i and j were invalid.
-2621	lip_setup_bimond	trouble allocating temporary workspace.
-2625	lip_setup_bimond	error return from PBHpm.
-2627	lip_setup_bimond	error return from lip_coeff_biherm.
-2701	lip_setup_biquad	nxin < 3.
-2702	lip_setup_biquad	nyin < 3.
-2703	lip_setup_biquad	both nxin and nyin were invalid.
-3000	lip_evalu_bivar	Input interp is NULL (un-allocated?).
-3001	lip_evalu_bivar	Input interp setup_type does not have one of the allowed values.
-3002	lip_evalu_bivar	Input interp contains one-dimensional data.
-3003	lip_evalu_bivar	Input int_type is not a legitimate LIP_meth.
-3004	lip_evalu_bivar	Input npts is not positive.
-3005	lip_evalu_bivar	Could not allocate space for index array.
-3007	lip_evalu_bivar	Input int_type=0, but interp int_type is not a legitimate LIP_meth.
-3008	lip_evalu_bivar	Input int_type>0, differs from interp int_type.
-3010	lip_evalu_bivar	Non-positive return value from lip_boxes. (Should never occur.)
-3100	lip_evalu_box	Input interp is NULL (un-allocated?).
-3101	lip_evalu_box	Input interp setup_type does not have one of the allowed values.
-3102	lip_evalu_box	Input interp contains one-dimensional data.
-3103	lip_evalu_box	Bad input int_type: not a legitimate LIP_meth or incompatible interp setup_type or int_type.
-3104	lip_evalu_box	Input npts is not positive.

-3203	lip_evalu_bilinear	interp setup_type indicates neither interpolation coefficients nor data available.
-3204	lip_evalu_bilinear	interp int_type is incompatible with LIP_LIN.
-3303	lip_evalu_bicubic	interp setup_type indicates neither interpolation coefficients nor data are available.
-3304	lip_evalu_bicubic	interp int_type incompatible with LIP_CUB (Only when setup_type=LSU_COEF.).
-3305	lip_evalu_bicubic	trouble allocating temporary storage.
-3306	lip_evalu_bicubic	error return from lip_deriv_est_box.
-3322	lip_evalu_bicubic	coefficient calculation failed due to singular matrix (should never happen with this version)
-3353	lip_evalu_bicubic2p	interp setup_type indicates neither interpolation coefficients nor data are available.
-3354	lip_evalu_bicubic2p	interp int_type incompatible with LIP_CUBM.
-3403	lip_evalu_biherm	interp setup_type indicates neither interpolation coefficients nor data available.
-3404	lip_evalu_biherm	interp int_type incompatible with LIP_HERM. (Only when setup_type=LSU_COEF.)
-3405	lip_evalu_biherm	trouble allocating temporary storage.
-3406	lip_evalu_biherm	error return from lip_deriv_est_box.
-3453	lip_evalu_bimond	interp setup_type indicates neither interpolation coefficients nor data available.
-3454	lip_evalu_bimond	interp int_type incompatible with LIP_MONO. (Only when setup_type=LSU_COEF.)
-3455	lip_evalu_bimond	trouble allocating temporary storage.
-3456	lip_evalu_bimond	error return from PBHpm.
-3503	lip_evalu_biquad	interp int_type or interp setup_type is incompatible with int_type=LIP_QUAD.
-3603	lip_evalu_birat	interp setup_type indicates data is not available.
-3605	lip_evalu_birat	trouble allocating temporary storage.
-4000	lip_evalu_univar	Input interp is NULL (un-allocated?).
-4001	lip_evalu_univar	Input interp setup_type does not have one of the allowed values.
-4002	lip_evalu_univar	Input interp contains two-dimensional data.
-4003	lip_evalu_univar	Input int_type is not a legitimate LIP_meth.
-4004	lip_evalu_univar	Input npts is not positive.
-4005	lip_evalu_univar	Could not allocate space for index array.
-4010	lip_evalu_univar	Non-positive return value from lip_cells. (Should never occur.)
-4100	lip_evalu_cell	Input interp is NULL (un-allocated?).
-4101	lip_evalu_cell	Input interp setup_type does not have one of the allowed values.
-4102	lip_evalu_cell	Input interp contains two-dimensional data.
-4103	lip_evalu_cell	Bad input int_type: not a legitimate LIP_meth or incompatible interp setup_type or int_type.
-4104	lip_evalu_cell	Input npts is not positive.
-4203	lip_evalu_linear	interp setup_type indicates neither data nor interpolation coefficients are available.
-4204	lip_evalu_linear	interp int_type is incompatible with LIP_LIN. (Only when setup_type=LSU_COEF.)
-4303	lip_evalu_cubic	interp setup_type indicates neither interpolation coefficients nor derivatives are available.
-4304	lip_evalu_cubic	interp int_type is incompatible with LIP_CUB. (Only if setup_type=LSU_COEF.)
-4310	lip_evalu_cubic	Trouble allocating temporary storage. (Only if setup_type=LSU_DATA.)
-4403	lip_evalu_hermit	interp setup_type indicates neither interpola-

```
                                 tion coefficients nor derivatives are available
-4404    lip_evalu_hermit    interp int_type is incompatible with LIP_HERM or
                             LIP_MONO.  (Only if setup_type=LSU_COEF.)
-4405    lip_evalu_hermit    interp int_type=LIP_MONO and setup_type=LSU_DATA.
-4410    lip_evalu_hermit    Trouble allocating temporary storage.
                             (Only if setup_type=LSU_DATA.)
-4603    lip_evalu_rat       interp setup_type indicates data is not
                             available.
-5000    (various)           Input interp is NULL (un-allocated?).
-5800    lip_set_int_type    Input interp is NULL (un-allocated?).
-5801    lip_set_int_type    Input interp is uninitialized or has no data.
-5802    lip_set_int_type    Input int_type is invalid for this data set.
-6500    lip_cubic_der2_cell Illegal value of icell.
-6605    lip_deriv_est_box   trouble allocating temporary storage.
-6610-m  lip_deriv_est_box   m>0 "Illegal value of icell" error returns
                             were received from lip_cubic_der2_cell, which
                             can only happen if ibox and/or jbox is bad.
-7000    lip_inverse_vals    Input interp is NULL (un-allocated?).
-7001    lip_inverse_vals    Input interp setup_type does not have one of
                             the allowed values.
-7002    lip_inverse_vals    Input interp contains one-dimensional data.
-7003    lip_inverse_vals    Input int_type is not a legitimate LIP_meth.
-7004    lip_inverse_vals    Input npts is not positive.
-7005    lip_inverse_vals    Could not allocate space for index array.
-7007    lip_inverse_vals    Input int_type=0, but interp int_type is not a
                             legitimate LIP_meth.
-7009    lip_inverse_vals    Invalid value of inv_mode.
-7010    lip_inverse_vals    Error return from lip_inv_coeff.
-7011    lip_inverse_vals    Error return from lip_inv_general.
-7012    lip_inverse_vals    Error return from lip_inv_partial.
-7301    lip_inv_coeff       Input interp setup_type is not LSU_COEF.
-7302    lip_inv_coeff       Input interp int_type is invalid.
-7303    lip_inv_coeff       Input int_type does not agree with that in
                             interp.
-7304    lip_inv_coeff       Coefficient inversion is not supported for
                             int_type.
-7501    lip_inv_general     Error return evaluating the interpolant.
-7502    lip_inv_general     Reached a theoretically impossible branch in
                             the binary search algorithm.
-7701    lip_inv_partial     setup_type in interp is LIP_COEFF.  (Should
                             use lip_inv_coeff, instead.)
-7702    lip_inv_partial     Partial setup is not supported for int_type.
-7703    lip_inv_partial     Input int_type does not agree with that in
                             interp.
-7721    lip_inv_bicub       Trouble allocating temporary storage space.
-7722    lip_inv_bicub       Coefficient matrix is singular. (Should not
                             happen.)
-7741    lip_inv_biher       Trouble allocating temporary storage space.
-7761    lip_inv_bimon       Trouble allocating temporary storage space.
-7761    lip_inv_jlim        Trouble allocating temporary storage space.
-7765    lip_coef_strip_bimon Trouble allocating temporary storage.
-7766    lip_coef_strip_bimon Error return from PBHpm.
```