UCRL-MA-108965 Rev. 1

PML User's Manual

for
PML Version 2.0

Stewart A. Brown
Dennis Braddy

Lawrence Livermore National Laboratory

Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Table of Contents

1.0 Introduction

There are many mathematics libraries currently enjoying widespread use. At the time that PML was begun, however, there were very few written in C and fewer still written with portability in mind. The temptation has all too often been to program to a specific hardware platform and forget portability. More recently some very good work has been published by Press, Flannery, Teulkovsky, and Vetterling, Numerical Recipes in C, which is highly portable. The reason for writing PML hasn't changed in spite this. No one library is apt to have everything that a given application needs and there are other considerations such as interfaces and related functionality to drive the development of a math library.

PML (Portable Mathematics Library) was developed initially as a repository for any routines of a mathematical nature in what has become the PACT system. More recently, I have tried to systematize the library and organize it for future growth.

The library has two fundamental parts. First are the many pure math routines such as equation solvers. Second are the structure definitions and some related functions. The structures which PML supports are provided to aid C programmers in organizing their mathematical computations in a mathematical way. That is, they are encouraged to give primary consideration to the mathematics not the implementation of the mathematics. This also is intended to add to the mathematical rigor of simulations and promote the use of some of the more abstract but critical ideas of mathematics by giving application developers concrete representations of abstract structures.

The data structures for PML are presented first since they support most of the library. A summary of the mathematical functions comes next. A set of examples completes this manual.

I would like to acknowledge the work of Charles F. McMillan in PML. He devised the original matrix structure and wrote many of the routines to manipulate them. The final form that they take in PML is very close to his original conception.

2.0 Data Structures

PML currently supports the following structures: complex, matrix; set; mapping; and field. The complex structure and its associated functions provide a rudimentary facility to do arithmetic with complex numbers. The matrix structure contains “all” of the information needed to deal with first and second rank tensors (i.e. vectors and matrices). The set structure is intended to describe a discrete set of elements. It includes information about the type of elements, the dimensionality, topology, and so on. Once the notion of a set has been built up, it is natural to proceed to discuss mappings of sets. This is obviously useful for such tasks as interpolations between computational meshes and visualization. The mapping structure includes a domain and range set and information describing the mapping between them. To support the mapping ideas, it is necessary to consider whether the sets involved are algebras, rings, fields, or whatever. To facilitate this thinking, a field structure is defined to provide the set of operations which can be carried out on the elements of sets. This also makes the set and mapping structures very general (which it should be) and still reasonably efficient.

2.1 Complex

The complex structure is the natural representation of a complex double precision number.

```
struct s_complex
   {double real;
    double imag;};
typedef struct s_complex complex;
```

The real part of the complex number is kept in the member **real** and the imaginary part of the complex number is kept in the member **imag**.

2.2 PM_matrix

The PM_matrix structure is a useful way to group a contiguous block of memory together with two integers which describe the logical shape of the space.

```
struct s_PM_matrix
   {int nrow;
    int ncol;
    REAL *array;};
typedef struct s_PM_matrix PM_matrix;
```

The **nrow** member is the number of rows of the matrix and the **ncol** member is the number of columns of the matrix. The **array** member is a pointer to the actual space. The type declaration **REAL** is set in the PACT system to be either **float** or **double**. Typically it is set to **double**, and **float** is used mainly on the smaller machines to conserve memory.

In this implementation, matrices are assumed to have a one based indexing scheme. That is, all matrix indices go from 1 to *ncol* or from 1 to *nrow*.

When operating on matrices it is necessary only to pass the matrix as an argument since the number of rows and columns is contained in the structure. A vector is taken to be simply a matrix with either one row or one column.

2.3 Set

The **PM_set** structure is an attempt to describe a usefully general mathematical set for such purposes as interpolation and visualization. It encapsulates a collection of information about a discrete set of elements of arbitrary type.

```
struct s_PM_set
   {char *name;
    char *element_type;
    int dimension;
    int *max_index;
    int dimension_elem;
    long n_elements;
    byte *elements;
    char *es_type;
    byte *extrema;
    byte *scales;
    PM_field *opers;
    REAL *metric;
    char *symmetry_type;
    byte *symmetry;
    char *topology_type;
    byte *topology;
    char *info_type;
    byte *info;};
typedef struct s_PM_set PM_set;
```

The **PM_set** members are: **name**, every set has a unique identifier; **element_type,** the type of element in the set; **dimension**, the dimension of the set; **max_index**, an array of the lengths along each dimension (it is *dimension* long); **dimension_elem**, the dimensionality of the elements of the set; n**_elements**, the number of elements in the set; **elements**, a pointer to the actual elements of the set; **es_type**, the type of the *extrema* and the *scales* arrays (typically, *element_type* is **double **** while *es_type* is **double ***); **extrema**, an array of minimum and maximum values for each component (2**dimension_elem* long); **scales**, an array of scales for each dimension; **opers**, specifies the field for the set; **metric**, an appropriate array of metric coefficients for the set if it is a metric set; s**ymmetry_type**, an ASCII string naming the type of the data in the **symmetry** member; **symmetry**, a pointer to data describing symmetries of the set, if any exist; **topology_type**, an ASCII string naming the type of the data in the **topology** member; **topology**, a pointer to data describing the connectivity of the elements of the set; **opers**, a collection of functions describing defined operations on elements of the set; **info_type**, an ASCII string naming the type of the data in the *info* member; and **info**, a pointer to data describing any additional information about the set.

In order to be general and useful, many types of information are described by two members of the **PM_set**. For example, an application may have a mesh (handled as a set) with an array of integers to define the connectivity. The **topology** member should point to that array and the **topology_type** member should be set to **integer** or **int**. The application would have easy access to the connectivity and another application which might work with sets of other kinds could be programmed to distinguish between the topological information associated with these mesh sets and that of other sets it knows about.

2.3.1 Example of building a set

This example is taken from the source for PML and illustrates how a set can be built. The routine uses the SCORE functions **MAKE**, **MAKE_N**, **SC_arrlen**, and **SC_strsave** as well as the SCORE variable **SC_DOUBLE_S**.

```
#include "pml.h"

main()
   {int i, k, l, kmax, lmax, kxl;
    REAL *x, *y, *u, *v, r, t;
    REAL xmin, xmax, ymin, ymax, km, lm;
    PM_set *domain, *range;

/* set up data */
    kmax      = 20;
    lmax      = 20;
    xmin      = -5.0;
    xmax      = 5.0;
    ymin      = -5.0;
    ymax      = 5.0;

    kxl = kmax*lmax;
    x   = MAKE_N(REAL, kxl);
    y   = MAKE_N(REAL, kxl);
    u   = MAKE_N(REAL, kxl);
    v   = MAKE_N(REAL, kxl);

    km = 2.0*PI/((double) (kmax - 1));
    lm = 2.0*PI/((double) (lmax - 1));
    for (k = 0; k < kmax; k++)
        for (l = 0; l < lmax; l++)
            {i    = l*kmax + k;
             x[i] = k*km;
             y[i] = l*lm;
             u[i] = sin(k*km);
             v[i] = sin(l*lm);};

/* build the domain set */
    domain = PM_make_set("{x, y}", SC_REAL_S, FALSE,
 2, kmax, lmax, 2, x, y);
```

```
/* build the range set */
    range = PM_make_set("{u, v}", SC_REAL_S, FALSE,
2, kmax, lmax, 2, u, v);

                .
                .
                .

    return(0);}
```

2.4 Mapping

Once the **PM_set** structure is there to organize data into some semblance of a mathematical set, it is quite natural to express objects such as computational meshes as sets. In fact, considering the needs of numerical simulation codes and visualization codes, the concepts of domain sets and range sets arise and lead to the need for a structure to characterize a mathematical mapping.

```
struct s_PM_mapping
   {char *name;
    char *category;
    PM_set *domain;
    PM_set *range;
    char *map_type;
    byte *map;
    in file_type;
    byte *file_info;
    char *file;
    struct s_PM_mapping *next;};
typedef struct s_PM_mapping PM_mapping;
```

The members of the **PM_mapping** are: **name**, a name for the mapping; **category**, the category to which the mapping belongs (e.g., Logical-Rectangular, Arbitrarily-Connected); **domain**, the mapping's domain; **range**, (a subset of) the image of domain under *map*; **map_type**, an ASCII string naming the type of the data pointed to by *map*; **map**, a pointer to data describing the mapping between *domain* and *range*; **file_type**, file type **ASCII**, **BINARY**, or **PDB**; **file_info**, file information (cast to some struct with information); **file**, the file name for the mapping; and **next**; the next **PM_mapping** in the chain.

At this point, many of these ideas are evolving. To insulate the developer somewhat against these changes, the **map** member was used. It allows different structures to be used to capture the description of the mapping from *domain* to *range*. For example, to handle the case when one has a domain and a function, the range could be NULL. Another potential issue to handle is whether the mapping is a bijection, surjection, or injection.

A very immediate issue that arises instantly is the centering of the range set relative to the domain set. For example, the domain might be a mesh with coordinates defining node centers while the range set might be a zone centered quantity. One small structure is provided to address this issue:

```
struct s_PM_map_info
   {char *name;
    int centering;};
typedef struct s_PM_map_info PM_map_info;
```

The **PM_map_info** structure only has a **name** member and an integer member, **centering**, *t*o specify the relative centering as being one of: **Z_CENT** for zone centered; **N_CENT** for node centered; **F_CENT** for face centered; **E_CENT** for edge centered; or **U_CENT** for uncentered.

2.5 Connectivity

To support general computational meshes PML has a model for arbitrarily connected meshes which is independent of dimensionality and a priori assumptions about shapes of zones or elements. We give a complete discussion below including definitions of terms. Hopefully this will suffice to give a useful account of this topic.

2.5.1 Concepts

The goal is to capture non-trivial topological information describing computational meshes which is efficient in space usage and amenable to fast algorithms for visualization rendering and analytical operations.

The representation of the connectivity scheme must accomodate a heirarchy of information. At the bottom of the heirarchy is the minimal information needed by a minimal connectivity scheme. At the top is a maximal set of information which is computable from the minimal information but is kept in memory rather than recomputing it.

2.5.2 Definitions

1. Computational mesh or mesh - a collection of discrete points in some vector space along with the specification of relationships between points especially the neighbor relationship.
2. Connectivity - the pattern of specifications of node neighbors.
3. N-cell - an N dimensional volume unit. Example: 3 dimensional mesh. A zone is a 3-cell, a face is a 2-cell, an edge is a 1-cell, and a node is a 0-cell.
4. N-Center - the center of an N-cell is defined by some weighted average of the real nodes which imply the N-cell. A 0-Center corresponds to a real node.
5. Boundary - the boundary of an N-cell is a set of (N-1)-cells such that the boundary of the boundary is 0. Although each (N-1)-cell may have a boundary, the set of them may not. This is a closure property. Each boundary (N-1)-cell has an orientation. The outward normal is the canonical orientation.
6. Zone - for a mesh in an N dimensional vector space a zone is the smallest N dimensional volume unit which is delineated by real nodes. In an N dimensional mesh, a zone is an N-cell.
7. Node - for a mesh in an N dimensional vector space a node is a single point in the N dimensional vector space. A node is always a 0-cell. The collection of nodes is one of the defining elements of the mesh.
8. Real Node - a node in the mesh which is explicitly specified and is a part of the connectivity of the mesh.
9. Virtual Node - a node which is implicitly defined by the mesh connectivity scheme. For example, the points defining the zone centers may be virtual nodes.

10. Decomposition - an N-cell which is implied by a set of real nodes can be expressed as a sum of N-cells which are implied by the set of real nodes and at least one virtual node.

11. Irreducible Decomposition - an N-cell has a special, unique decomposition which is the sum of N-cells each of which has: 1 0-Center, 1 1-Center, ..., 1 N-Center in their boundary and which collectively cover the N-cell.

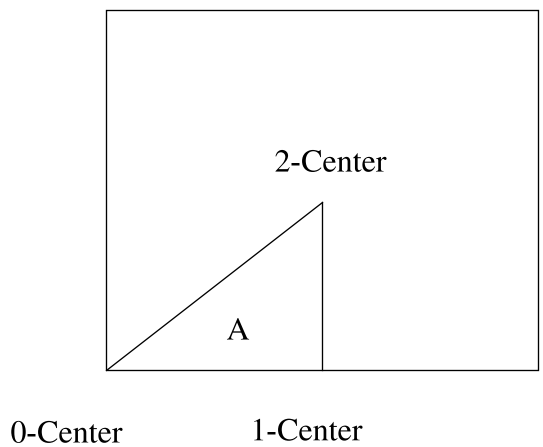

FIGURE 1. The cell A is one cell in the irreducible decomposition of the rectangular 2-Cell shown

2.5.3 Axioms

- The mimimum information necessary to describe a computational mesh is: 1) a list of nodes; and 2) for each node a list of its neighboring nodes.
- The information that will typically be available in a mesh description is: 1) a list of nodes; 2) a list of zones; and 3) for each node a list of its neighboring nodes.

2.5.4 Representation Requirements

To represent an N-cell obviously depends on how much information you want to maintain.

- Minimal:

  Boundary specification (necessary and sufficient)

- Maximal:

  Center node

(N+1)-cell for which this N-cell is a boundary segment

Neighboring (N+1)-cell

Decomposition specification

Irreducible Decomposition specification

A flexible approach is taken by representing a cell as a block of contiguous integers in an array of cells. This means that an array of cells is in effect a two dimensional array of integers. The fastest changing dimension is bounded by the number of parameters used in representing the cells and the slowest changing dimension is the cell number.

PML defines the following constants to name the cell description parameters

```
#define BND_CELL_MIN  0
#define BND_CELL_MAX  1
#define OPPOSITE_CELL 2
#define PARENT_CELL   3
#define NGB_CELL      4
#define CENTER_CELL   5
#define DEC_CELL_MIN  6
#define DEC_CELL_MAX  7
```

2.5.5 Representation of Connectivity

The PM_mesh_topology structure attempts to describe in complete generality a mesh of points and their neighbors as a heirarchy in which the highest dimensional volumes sit at the top and the line segment descriptions sit at the bottom.

At each level in the heirarchy an N-cell is described by a contiguous block of (N-1)-cell boundary sections (next lower down in the heirarchy).

Assuming the dimensionality to be n the heirarchy goes like:

boundaries[n]   the description for the n-dim surfaces

boundaries[n-1]   the description for the (n-1)-dim surfaces

.

.

.

boundaries[1]   the description for the 1-dim surfaces, i.e. line segments whose endpoints are the node indices

boundaries[0]   a mimimal description for the nodes

```
struct s_PM_mesh_topology
   {int n_dimensions;        /* number of dimensions */
    int *n_bound_params;      /* number of bound params at each level */
    int *n_cells;            /* number of cells at each level */
    long **boundaries;};     /* boundary info array at each level */
```

typedef struct s_PM_mesh_topology PM_mesh_topology;

For example, in a 3 dimensional problem, the “zones” at the top of the hierarchy are 3-cells. Each “zone” has some number of “faces” or 2-cells bounding it. Each “face” has some number of “edges” or 1-cells bounding it. Implicitly, each “edge” has two points or 0-cells bounding it. Then boundaries[3] would contain the indices into boundaries[2] of boundary faces and, if that level of information is kept, indices of other zones in boundaries[3] which are neighbors.

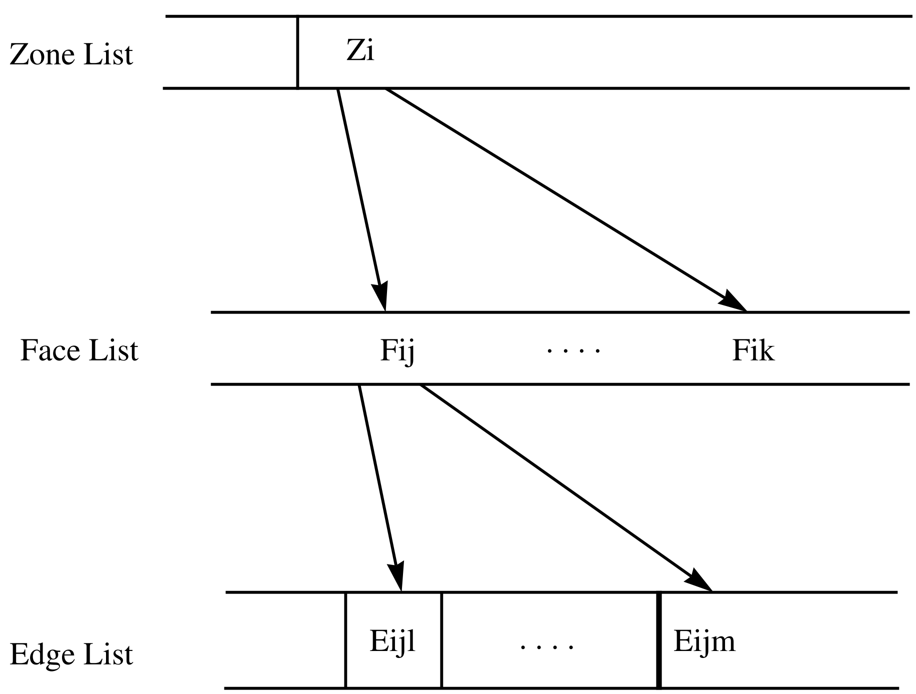

FIGURE 2. Diagram of hierarchy for 3 dimensional mesh connectivity

2.6 Field

Depending on which operations are defined on a set of elements (in addition to other properties), one has a vector space, an algebra, a ring, a group, a field, or variations on that same theme. As a practical matter, given a set of elements, applications need to know how to combine them (add, subtract, etc.). The **PM_field** structure provides a way to make those connections in a generic way.

```
struct s_PM_field
   {byte (*add)();
    byte (*sub)();
    byte (*scalar_mult)();
    byte (*mult)();
    byte (*div)();};
typedef struct s_PM_field PM_field;
```

3.0 Summary of the PML C API

Here is a brief summary of the routines in PML. They are broken down by category as: matrix manipulation routines; other equation solvers; geometry routines; mathematical functions; constructors and destructors; field operators; and sorting routines.

3.1 MATRIX Manipulation Routines

PM_matrix *PM_create(int nrow, int ncol)

Create and return a new PM_matrix with *nrow* rows and *ncol* columns. If space cannot be allocated, NULL is returned.

PM_matrix *PM_decompose(PM_matrix *a, int *ip, int flag)

Perform an LU decomposition on the PM_matrix *a*. The pivoting information is returned in *ip*. The space for *ip* must be passed into this function. If *flag* is TRUE a new matrix is allocated to hold the decomposed matrix. Otherwise the original matrix is overwritten by the decomposed matrix. NULL is returned if the routine fails.

PM_matrix *PM_decb(int n, int ml, int mu, PM_matrix *b, int *ip)

Perform an LU decomposition on the banded matrix *b*. The arguments *n*, *ml*, and *mu* are the order of the matrix, the lower band width, and the upper band width respectively. The pivoting information is returned in *ip*. The space for *ip* must be passed into the routine. This routine replaces the contents of *b* with the decomposed matrix and returns a pointer to *b* if successful. If the routine fails it returns NULL.

int PM_destroy(PM_matrix *m)

Release the storage associated with the matrix *m*. Both the array and the matrix structure are freed. TRUE is returned if successful otherwise FALSE is returned.

PM_element(PM_matrix *m, int row, int column)

This macro accesses the matrix element, *m*(*row*, *column*). Its value can either be read or set.

PM_matrix *PM_inverse(PM_matrix *a)

The matrix *a* is inverted via an LU decomposition. A new space is allocated to hold the inverse. If the inversion is successful the inverse matrix is returned otherwise NULL is returned.

int PM_inv_times_ds(PM_matrix *b, PM_matrix *c, PM_matrix *t, int *ip)

Using an LU decomposed matrix *b* compute $(b^{-1}).c$ and return it in *t*. *T*he pivoting information from the decomposition is supplied in *ip*.

PM_matrix *PM_lower(PM_matrix *a)

Create and fill a matrix consisting of the sub-diagonal part of the input matrix *a*. The new matrix is returned.

PM_matrix *PM_minus(PM_matrix *a, PM_matrix *b)

Create and fill a matrix consisting of the difference, *a*-*b*. The new matrix is returned.

PM_matrix *PM_minus_s(PM_matrix *a, PM_matrix *b, PM_matrix *c)

Compute the difference, *b*-*c*. The result is returned in the matrix *a*.

PM_matrix *PM_plus(PM_matrix *a, PM_matrix *b)

Create and fill a matrix consisting of the sum, *a*+*b*. The new matrix is returned.

PM_matrix *PM_plus_s(PM_matrix *a, PM_matrix *b, PM_matrix *c)

Compute the sum, *b*+*c*. The result is returned in the matrix *a*.

PM_matrix *PM_print(PM_matrix *a)

Print the matrix *a* to stdout. The matrix *a* is returned

PM_matrix *PM_sol(PM_matrix *ul, PM_matrix *b, int *ip, int flag)

Using the LU decomposed matrix *ul* and the pivoting information in *ip* as computed by **PM_decompose**, solve for the solution to the equation $A.x = b$. If *flag* is TRUE a new matrix is allocated to contain the solution and is returned. If *flag* is FALSE the solution is returned in *b* and a pointer to *b* is the return value of the function.

PM_matrix *PM_solb(int n, int ml, int mu, PM_matrix *b, PM_matrix *y, int *ip)

Using the LU decomposed matrix *b* and the pivoting information in *ip* as computed by **PM_decb**, solve for the solution to the equation $A.x = y$ where *A* is a banded matrix. The arguments *n*, *ml*, and *mu* are the order of the matrix, the lower band width, and the upper band width respectively. The solution is returned in *y* and a pointer to *y* is the return value of the function.

PM_matrix *PM_solve(PM_matrix *a, PM_matrix *b)

The equation $a.x = b$ is solved and the result returned in *b*.

PM_matrix *PM_times(PM_matrix *a, PM_matrix *b)

Create and fill a matrix consisting of the product, *a.b*. The new matrix is returned.

PM_matrix *PM_times_s(PM_matrix *a, PM_matrix *b, PM_matrix *c)

Compute the product, *b.c*. The result is returned in the matrix *a*.

PM_matrix *PM_transpose(PM_matrix *a)

Compute the transpose of the matrix *a*. A new matrix is created to contain the transpose and is returned.

PM_matrix *PM_upper(PM_matrix *a)

Create and fill a matrix consisting of the super-diagonal part of the input matrix *a*. The new matrix is returned.

3.2 Other Equation Solvers

The matrix solvers presented here are for sparse matrices. Sparse matrices do not fit very naturally into the matrix structure discussed above. Hence the solvers here do not use matrix structures.

int PM_block_tridi(REAL *a, REAL *b, REAL *c, REAL *u, int ns, int nb)

Solve the equation $M.x = u$ where M is block tridiagonal. There are *nb* blocks each *ns* squared in size. The matrices *a*, *b*, and *c* are the sub-diagonal, diagonal, and super-diagonal parts respectively. They are (*ns***ns*)**nb* long and both the *a*[0] and *c*[*nb*-1] blocks are unused since the off-diagonal parts are one block shorter than the diagonal. The result is returned in *u*. All vectors are passed into the routine which returns TRUE if successful and FALSE otherwise.

**REAL PM_iccg(int km, int lm, double eps, int ks, int maxit, REAL *a0,
REAL *a1, REAL *b0, REAL *b1, REAL *bm1,
REAL *x, REAL *y)**

Use the ICCG method to solve the system of equations, $M.x$ = y. The array M is symmetric and positive definite. Its components are passed in as the arrays: *a0*, *a1*, *b0*, *b1*, and *bm1*. The known values are passed in via the *y* array. The input arrays are all twice as long as needed to specify the problem; the second half is workspace. That is, the input arrays should all have dimension at least 2*(*km***lm*).The matrix M to be solved is symmetric, and has the structure:

$$
\begin{array}{cccccccccc}
a0_1 & & & & & & & & & \\
a1_1 & a0_2 & & & & & & & & \\
 & a1_2 & a0_3 & & & & & & & \\
 & & . & & & & & & & \\
 & & & . & & & & & & \\
 & & & & . & & & & & \\
 & & & & a1_{km-2} & a0_{km-1} & & & & \\
 & & & & & a1_{km-1} & a0_{km} & & & \\
b0_1 & bm1_1 & & & & & & a0_{km+1} & & \\
b1_1 & b0_2 & bm1_2 & & & & & a1_{km+1} & a0_{km+2} & \\
 & b1_2 & b0_3 & bm1_3 & & & & & a1_{km+2} & a0_{km+3} \\
 & . & & & & & & & . &
\end{array}
$$

Note that elements $a1_{km}$ etc. are zero due to the block structure. The parameters *km* and *lm* are the dimensions of the underlying computational mesh. The index $0 <= k < km$ is the rapidly varying index of the mesh array. The convergence criterion is controlled by the dimensionless parameter *eps*. When the solution changes by less than *eps*, convergence is assumed. When PML runs on a vector machine, the incomplete Cholesky decomposition is done by cyclic reduction. To control the number of levels of cyclic reduction the *ks* parameter is supplied. One more level than specified will be performed. The minimum possible value of *ks* is 0, and the maximum possible value of *ks* is *kp*-1, where *kp* is the highest power of 2 with $2^{kp} <= lm$. A good choice for *ks* is 4 for "most" problems. On a scalar machine this parameter is ignored. The number of iterations in the conjugate gradient step is controlled by the parameter *maxit*. The return value, *ret*, is the actual number of conjugate gradient iterations plus the norm of the residual. The value is multiplied by -1 if certain exceptional conditions arise. Failure of the algorithm then is indicated if any of the following is true:

$$ret < 0$$

$$maxit <= \mathrm{int}(ret)$$

$$eps < \mathrm{frac}(ret)$$

The *x* array contains the solution on return.

int PM_tridi(REAL *a, REAL *b, REAL *c, REAL *r, REAL *u, int n)

Solve the equation $M.u = r$ where M is tridiagonal. The matrices *a*, *b*, and *c* are the sub-diagonal, diagonal, and super-diagonal respectively and both *a*[0] and *c*[*n*-1] are unused since the off-diagonal vectors are one element shorter than the diagonal. The number of equations is *n*. The result is returned in *u*. All vectors are passed into the routine which returns TRUE if successful and FALSE otherwise.

3.3 Geometry Routines

REAL PM_dot(REAL *x, REAL *y, int n)

Take the inner product of two vectors of length *n* and return the result.

int PM_cross(double x1, double y1, double x2, double y2, double x3, double y3, double x4, double y4, double *px0, double *py0)

Return the intersection point of the line segment defined by (X_1 - X_2) and the ray defined by (X_3 - X_4). The point X_3 is the end point of the ray. Each vector X_i has the components X_i = (*xi*, *yi*). The intersection point is returned in (*px0*, *py0*). If the ray does not cross the segment (not the line, the segment!) FALSE is returned and the vector (**HUGE**, **HUGE**) is passed back in *px0* and *py0*. If ray does cross TRUE is returned. **HUGE** is a #define'd constant signifying a large floating point number

int PM_cross_seg(double x1, double y1, double x2, double y2, double x3, double y3, double x4, double y4, double *px0, double *py0)

Lik PM_cross this function will return the intersection point of the lines implied by the two segments given. However, it returns TRUE if and only if the segments cross.

int PM_cross_line_plane(double x1, double y1, double z1, double x2, double y2, double z2, REAL *px, REAL *py, REAL *pz, REAL *px0, REAL *py0, REAL *pz0, int line)

This function tests for the intersection of lines and a plane. The value of *line* may be 0, 1, 2 for segment, ray, or line. The line is defined by the vectors: X1 and X2 the plane is defined by the vectors: X3, X4, X5. For rays X1 is the terminations of the ray (i.e. tail of vector) and X2 is X1 - dX (dX defines the direction of the ray). Vectors are defined as X = (x, y). TRUE is returned if and only if the segment, ray, or line crosses the plan.

int PM_colinear_2d(REAL *px, REAL *py, int n)

Return TRUE if and only if the *n* points from the *px* and *py* arrays are colinear in 2 dimensions.

int PM_colinear_3d(REAL *px, REAL *py, REAL *pz, int n)

Return TRUE if and only if the *n* points from the *px, py*, and *pz* arrays are colinear in 3 dimensions.

int PM_contains_2d(double x, double y, REAL *px, REAL *py, int n)

Return TRUE if and only if the point (*x, y*) is contained in the plane bounded by the polygon defined by the *n* points in the *px* and *py* arrays.

int PM_contains_3d(double x, double y, double z, REAL *px, REAL *py, REAL *pz, int bnd)

Return TRUE if and only if the point (*x, y, z*) is contained in the plane bounded by the polygon defined by the *n* points in the *px, py*, and *pz* arrays. If *bnd* it TRUE points on the boundary are included.

int PM_intersect_line_polygon(REAL *pxmn, REAL *pymn, REAL *pxmx, REAL *pymx, REAL *ppx, REAL *ppy, int np, int *pic)

This routine computes the intersection points of the line given by (*pxmn*, *pymn*) and (*pxmx*, *pymx*) and the given polygon as defined by the *np* points in *ppx* and *ppy*. The number of intersection points is returned in *pic* and the intersection points are returned in (*pxmn*, *pymn*) and (*pxmx*, *pymx*). FALSE is returned if the line segment is completely outside the polygon.

void PM_convex_hull(REAL *px, REAL *py, int nh, REAL **ppx, REAL **ppy, int *pnp)

This routine allocates and returns arrays (*ppx*, *ppy*) containing *pnp* points defining the convex polygon which contains the *nh* points specified in the *px* and *py* arrays.

void PM_nearest_point(REAL *px, REAL *py, int n, double xs, double ys, REAL *pxt, REAL *pyt, int *pi)

This routine finds the point in the curve defined by the *n* points in the *px* and *py* arrays which is nearest to the point (*xs*, *ys*). The point is returned in (*pxt*, *pyt*) and the index of that point in *pi*.

3.4 Mathematical Functions

double PM_fix(double value)

Return the integer part of *value* as a REAL number. Unlike the C library function **floor**, PM_fix(-2.3) = -2.0;

double PM_frac(double x)

Return the fractional part of *x* as a REAL number.

double PM_power(double x, int p)

Return x^p.

double PM_random(double x)

Return a random number in the range -1.0 to 1.0.

double PM_round(double x)

Round the number *x* to the nearest integer returned as a REAL number.

double PM_sgn(double value, double sign)

Return sign(*sign*)*abs(*value)*.

double PM_sign(double x)

Return the sign of *x* (-1.0 or 1.0) or 0 if $x = 0$.

double PM_sqr(double x)

Return x^2.

3.5 Constructors and Destructors

PM_set *PM_make_set(char *name, char *type, int cp, int nd, ...)

Allocate and return a **PM_set** with name *name* and data of type *type*. The *cp* flag specifies the data given will be copied if the value is TRUE otherwise the data given will be used directly. This can be very important because some applications require that the data in the set have been dynamically allocated by SCORE and the copy would guarantee that. Also this feature gives the application more control of data flow. The parameter *nd* is the dimension of the set. It is followed by *nd* integers representing the maximum value of an index for that dimension. The indices start at 0. Next comes the dimensionality of the elements of the set, *nde*. This is followed by *nde* arrays whose values are double precision floating point numbers. Each of these arrays must be as long as the product of the *nd* maximum indices.

This is not remotely the most general **PM_set** that can be represented, but it is one of the most frequently encountered types of set.

See the examples in the discussion of the **PM_set** structure.

void PM_rel_set(PM_set *set, int rel)

Release the space associated with the specified set. If *rel* is TRUE the data arrays will be released as well as the set structure.

PM_mapping *PM_make_mapping(char *name, char *cat, PM_set *domain, PM_set *range, int centering, PM_mapping *next)

Allocate and return a PM_mapping with name *name*, type *cat*, *domain*, *range*, and *centering*. The mapping type is one of “Logical-Rectangular” or “Arbitrarily-Connected”. If a linked list of mappings is desired the mappings to which the new one points is passed in *next*.

See the examples in the discussion of the PM_mapping structure.

void PM_rel_mapping(PM_mapping *f, int rld, int rlr)

Release the space associated with the specified mapping. If *rld* is TRUE the data arrays of the domain set will be released and if *rlr* is TRUE the data arrays of the range set will be released.

3.6 Field Operators

The following functions come in groups. They are provided to have at hand the functions for the **PM_field** structure which is used in connection with **PM_set**'s and **PM_mapping**'s. They are also useful in applications where a pointer to a function is needed and the function in question is addition, subtraction, multiplication, or division.

int PM_iplus(int x, int y)

int PM_iminus(int x, int y)

int PM_itimes(int x, int y)

int PM_idivide(int x, int y)

Add, subtract, multiply, or divide integer *x* and *y* values and return an integer result.

int PM_imodulo(int x, int y)

Return mod(*x, y*) as an integer value.

long PM_lplus(long x, long y)

long PM_lminus(long x, long y)

long PM_ltimes(long x, long y)

long PM_ldivide(long x, long y)

Add, subtract, multiply, or divide long integer *x* and *y* values and return a long integer result.

long PM_lmodulo(long x, long y)

Return mod(*x, y*) as a long integer value.

double PM_fplus(double x, double y)

double PM_fminus(double x, double y)

double PM_ftimes(double x, double y)

double PM_fdivide(double x, double y)

Add, subtract, multiply, or divide double precision floating point *x* and *y* values and return a double result.

3.7 Sorting Routines

int *PM_t_sort(int *in, int n_dep, int n_pts, int *ord)

This function implements the topological sort routine described in Knuth's The Art of Computer Programming, Vol 1. Input is an array of indices *in*, which is 2**n_dep* long, containing *n_dep* pairs of integers specifying the topology of the set to be sorted in terms of relations, $j < k$ ("*j* precedes *k*"). The number of points in the set is *n_pts*. If *ord* is NULL this routine will allocate space to return the ordering, otherwise it builds the ordering into *ord.* The array *ord* must be *n_pts* long if it is passed in. In any case, if successful, a pointer to the ordering array is returned. If there are loops in the topology specified by *in*, or there are other errors NULL is returned.

4.0 Summary of the PML FORTRAN API

Here is a brief summary of the routines in PML. They are broken down by category as: matrix manipulation routines; other equation solvers; geometry routines; mathematical functions; constructors and destructors; field operators; and sorting routines.

pmszft(integer n)

Returns the nearest integer power of 2 in the argument *n*. This is used to check the size of the output arrays passed into **pmrfft** or **pmcfft**. The actual array sizes must be one larger than this value (this is for the zero frequence component).

pmrfft(real outyr, real outyi, real outx, integer n, real inx, real iny, real xn, real xx, integer o)

Performs an FFT on *n* non-evenly spaced real data points.

The input arguments are: *n* an integer number of data points; *inx*, an array of *n* real x values; *iny*, an array of *n* real y values; *xn*, the minimum x value; *xx*, the maximum real value; and *o*, an integer flag signalling the ordering of the transformed data. If *o* is 1 then the FFT data is returned in increasing order of “frequency”. If *o* is 0 then the FFT data is returned with frequency starting at 0 and increasing followed by the most negative frequency increasing to 0.

The output arguments are: *outyr*, an array of real values constituting the real part of the transformed range; *outyi*, an array of real values constituting the imaginary part of the transformed range; and *outx*, an array of real values constituting the transformed domain (e.g. frequency). The calling routine must supply the output arrays. The size of the arrays is one greater than the value return by a call to **pmszft** with the size of the input arrays.

pmcfft(real outyr, real outyi, real outx, integer n, real inx, real inyr, real inyi, real xn, real xx, integer f, integer o)

Performs an FFT on *n* non-evenly spaced complex data points.

The input arguments are: *n* an integer number of data points; *inx*, an array of *n* real x values; *inyr*, an array of *n* real y values constituting the real part of the input range; *inyi*, an array of *n* real y values constituting the imaginary part of the input range; *xn*, the minimum x value; *xx*, the maximum real value; *f*, an integer 1 for an FFT and -1 for in inverse FFT; and *o*, an integer flag signalling the ordering of the transformed data. If *o* is 1 then the FFT data is returned in increasing order of “frequency”. If *o* is 0 then the FFT data is returned with frequency starting at 0 and increasing followed by the most negative frequency increasing to 0.

The output arguments are: *outyr*, an array of real values constituting the real part of the transformed range; *outyi*, an array of real values constituting the imaginary part of the transformed range; and *outx*, an array of real values constituting the transformed domain (e.g. frequency). The calling routine must supply the output arrays. The size of the arrays is one greater than the value return by a call to **pmszft** with the size of the input arrays.

integer pmbset(integer nn, char *name, integer nt, char *type, integer cp, integer nd, integer nde, integer maxes, integer topid, integer nextid)

Start building a PM_set. This function builds a set upto the elements of the set which are added with the **pmaset** routine. After all the components have been added a call to **pmeset** must be made to complete the set. An integer identifier for the set is returned. If its value is -1 then an error has occured and the set does not exist. The arguments are:

nn	number of characters in the set name
name	the name of the set
nt	number of characters in the set type
type	the type of the elements of the set
cp	if 1 the elements added with pmaset will be copied
nd	the number of dimensions of the set
nde	the number of dimensions of the set elements
maxes	an array of dimensions for logical rectangular sets
topid	an identifier for a PM_mesh_topology structure
nextid	an identifier for the next set in a linke list

If the set is logically rectangular then *maxes* should be an array containing the lengths of each dimension and *topid* should be 0. If the set is arbitrarily connected *maxes* should be 0 and *topid* should contain a valid mesh topology structure.

The return value is an integer which is used as an identifier in functions which require a reference to a set. If there is an error -1 will be returned.

integer pmaset(integer setid, integer component, void element)

Add component number *component* to the set. The data in *element* must be an array of numbers of the type defined in the set and with length matching the number of elements of the set. Typically this will be called *nde* times where *nde* is the number of dimensions of the set elements as supplied to **pmbset**. If the set was made with the copy flag on (1), then the values in *element* will be copied into a newly allocated space, otherwise the array *element* will be used directly. This point should always be very carefully considered since dynamic memory heaps can be corrupted if this is used improperly. The argument *setid* is the value returned from a call to **pmbset**.

integer pmeset(integer setid)

This completes the construction of the set begun with **pmbset** and **pmaset**. The argument *setid* is the value return from a call to **pmbset**.

integer pmmtop(integer nd, integer nc, integer bp, integer bnd)

Make a PM_mesh_topology structure from the given connectivity informations. You should be thoroughly familiar with the discussion of connectivity given earlier. The arguments are:

nd the number of dimensions

nc an array of numbers of cells of each dimension

bp an array of numbers of boundary parameters at each dimension

bnd an array containing the cell descriptions

The layout of *nc* has *nc(1)* being the number of 0-cells (nodes), *nc(2)* being the number of 1-cells (edges), *nc(3)* being the number of 2-cells (faces), and so on. The layout of *bp* has *bp(1)* is 0, *bp(2)* the number of parameters describing all 1-cells, *bp(3)* the number of parameters describing all 2-cells, and so on. The layout of *bnd* is as follows:

$bnd(n_1)$ thru $bnd(n_2-1)$ contains all of the 1-cell information

$bnd(n_2)$ thru $bnd(n_3-1)$ contains all of the 2-cell information

$bnd(n_3)$ thru $bnd(n_4-1)$ contains all of the 3-cell information

.

.

.

where

$n_i = 1 + nc(i)*bp(i)$

It is crucial to remember that the values in *bnd* are indeces which are 0 based! Be sure to review the section on connectivity.

5.0 EXAMPLES

To provide some examples of the use of these routines, the source code for some of the PML test programs is provided here. The first piece of code demonstrates the use of most of the matrix routines. This is followed by the ICCG test program and the test program for the topological sort routine. The routine to generate a **PM_set** was included in the discussion of the **PM_set** structure. Users may contact the author for further examples.

5.1 Examples of MATRIX Functions

```
#include "pml.h"

REAL a1[5][3] = {0, 0, 1, 1, 0, 1, 0, 1, 1,
                0.5, 0.5, 1, 0.7, 0.9, 1};
REAL b1[5][1] = {0.5, 1.4, .93589, 1.16795, 1.52230};

/* MAIN - a sample program */

main()
   {int i, j, count;
    int nrow = 5, ncol = 3;
    PM_matrix *m, *t, *b, *a, *c;

    m = PM_create(nrow, ncol);
    b = PM_create(nrow, 1);

    count = 0;
    for (i = 1; i <= nrow; i++, count++)
        {for (j = 1; j <= ncol; j++, count++)
             {PM_element(m, i, j) = count;};
         PM_element(b, i, 1) = count;};
    PM_print(m);

    m->array = (REAL *)a1;
    b->array = (REAL *)b1;

    PM_print(m);

    PM_print(b);

    PM_print(t = PM_transpose(m));

    PM_print(a = PM_times(t, m));

    PM_print(c = PM_times(t, b));

    PM_print(PM_solve(a, c));

    PM_print(t = PM_times(PM_transpose(m), m));
```

```
PM_print(c = PM_inverse(t));

PM_print(PM_times(c, t));

exit(0);}
```

5.2 Example of ICCG Routine

```
#include "pml.h"

#define KM     4
#define LM     3
#define KXL    24                                 /* 2*KM*LM */
#define KL     12                                   /* KM*LM */
#define EPS    1.0e-6
#define KS     4
#define MAXIT  100

/* MAIN - solve a Laplace equation */

main()
   {int i, j, k;
    REAL ret;
    REAL a0[KXL], a1[KXL], b0[KXL], b1[KXL], bm1[KXL],
         x[KXL], y[KXL];

/* solve a Laplacian in the l direction */
    for (i = 0; i < KXL; i++)
        {a0[i]  = 0.0;
         a1[i]  = 0.0;
         b0[i]  = 0.0;
         b1[i]  = 0.0;
         bm1[i] = 0.0;
         x[i]   = 0.0;
         y[i]   = 0.0;};

    for (i = KM; i < 2*KM; i++)
        {a0[i]  = 2.0;
         a1[i]  = -1.0;
         b0[i]  = 0.0;
         b1[i]  = 0.0;
         bm1[i] = 0.0;};

    a1[2*KM-1] = 0.0;
    for (i = 0; i < KM; i++)
        {k = KM + i;
         y[k] = (REAL) (i+1);};
```

```
    printf("\nProblem y :\n");
    for (j = 0; j < LM; j++)
        {printf("\nRow #%d: \n", j+1);
         for (i = 0; i < KM; i++)
             {k = j*KM + i;
              printf(" y(%2d, %2d) = %g ", j+1, i+1, y[k]);};};
    printf("\n");

    ret = PM_iccg(KM, LM, EPS, KS, MAXIT,
                  a0, a1, b0, b1, bm1, x, y);

    printf("\nSolution x (should be (4, 7, 8, 6)) : %g\n", ret);
    for (j = 0; j < LM; j++)
        {printf("\nRow #%d: \n", j+1);
         for (i = 0; i < KM; i++)
             {k = j*KM + i;
              printf(" x(%2d, %2d) = %g ",j+1, i+1, x[k]);};};
    printf("\n\n");

    SC_pause();

/* solve a Laplacian in the k direction */
    for (i = 0; i < KXL; i++)
        {a0[i]  = 0.0;
         a1[i]  = 0.0;
         b0[i]  = 0.0;
         b1[i]  = 0.0;
         bm1[i] = 0.0;
         x[i]   = 0.0;
         y[i]   = 0.0;};

    for (i = 1; i < KXL; i += LM)
        {a0[i]  = 2.0;
         a1[i]  = 0.0;
         b0[i]  = -1.0;
         b1[i]  = 0.0;
         bm1[i] = 0.0;};

    for (k = 1, i = 1; i < KXL; i += LM, k++)
        y[i] = (REAL) k;
```

```
printf("\nProblem y :\n");
for (j = 0; j < KM; j++)
    {printf("\nRow #%d: \n", j+1);
     for (i = 0; i < LM; i++)
         {k = j*LM + i;
          printf(" y(%2d, %2d) = %g ", j+1, i+1, y[k]);};};
printf("\n");

ret = PM_iccg(LM, KM, EPS, KS, MAXIT,
             a0, a1, b0, b1, bm1, x, y);

printf("\nSolution x (should be (4, 7, 8, 6)) : %g\n", ret);
for (j = 0; j < KM; j++)
    {printf("\nRow #%d: \n", j+1);
     for (i = 0; i < LM; i++)
         {k = j*LM + i;
          printf(" x(%2d, %2d) = %g ", j+1, i+1, x[k]);};};
printf("\n\n");

exit(0);}
```

5.3 Example of Topological Sort Routine

```
#include “pml.h”

#define NDEP 10
#define NPTS  9

main()
   {int i, rel[20], *ord;

    rel[0] = 9;
    rel[1] = 2;

    rel[2] = 3;
    rel[3] = 7;

    rel[4] = 7;
    rel[5] = 5;

    rel[6] = 5;
    rel[7] = 8;

    rel[8] = 8;
    rel[9] = 6;

    rel[10] = 4;
    rel[11] = 6;

    rel[12] = 1;
    rel[13] = 3;

    rel[14] = 7;
    rel[15] = 4;

    rel[16] = 9;
    rel[17] = 5;

    rel[18] = 2;
    rel[19] = 8;

    printf(“\nRelations List:\n”);
```

```
for (i = 0; i < 2*NDEP; i += 2)
    printf("%2d < %2d\n", rel[i], rel[i+1]);
printf("\n");

ord = PM_t_sort(rel, NDEP, NPTS, NULL);

printf("Partially ordered List:\n");
for (i = 0; i < NPTS; i++)
    printf("A(%2d) = %2d\n", i, ord[i]);

printf("\nThe correct order is:\n 1 9 3 2 7 4 5 8 6\n\n");

exit(0);}
```

6.0 Related Documentation

PML is a part of the PACT portable code development and visualization tool set. Interested readers may with to consult other PACT documents.

The list of PACT Documents is:

PACT User's Guide, UCRL-MA-112087

SCORE User's Manual, UCRL-MA-108976 Rev.1

PPC User's Manual UCRL-MA-108964 Rev.1

PML User's Manual, UCRL-MA-108965 Rev.1 (this document)

PDBLib User's Manual, M-270 Rev.2

PGS User's Manual, UCRL-MA-108966 Rev.1

PANACEA User's Manual, M-276 Rev.2

ULTRA II User's Manual, UCRL-MA-108967 Rev.1

PDBDiff User's Manual, UCRL-MA-108975 Rev.1

PDBView User's Manual, UCRL-MA-108968 Rev.1

SX User's Manual, UCRL-MA-112315

Additional reading:

Abramowitz, Stegun, Handbook of Mathematical Functions, Dover.

Knuth, D.E. The Art of Computer Programming, Vol I - III, Addison-Wesley.

Press, Flannery, Teukolsky, and Vetterling, Numerical Recipes in C, Cambridge Press.

R

S

T

U

V

Z