UCLA
COMPUTATIONAL AND APPLIED MATHEMATICS

Extensible PDE Solvers Package
Users Manual

Barry Smith

January 1995

CAM Report 95-4

Department of Mathematics
University of California, Los Angeles
Los Angeles, CA. 90024-1555

Extensible PDE Solvers Package
Users Manual

by

Barry Smith
Mathematics and Computer Science Division

September 1994

This work was supported in part by the Office of Scientific Computing, U.S. Department of Energy, and by ARO under subcontract number ORA 4466.04 Amendment 1 from The University of Tennessee, Knoxville, while the author was at the Department of Mathematics, University of California at Los Angeles.

This paper also appeared as: Argonne National Laboratory Report # ANL-94/40.

Contents

Extensible PDE Solvers Package
Users Manual

by

Barry Smith

Abstract

This manual describes the design and use of the Extensible PDE Solvers package for the solution of elliptic PDEs.

At this time, the package provides support for the solution of elliptic PDEs using either finite elements or finite differences in two or three dimensions on either structured or unstructured grids. The package is designed to be easily extended to new discretizations or classes of PDEs. Several classical direct and iterative methods, as well as several multigrid variants, may be used to solve the resulting linear systems.

The package is intended as a prototype, working implementation to demonstrate, and learn, how such packages may be organized. Many important features for particular problems are lacking; future versions of the package may add increased functionality. The software described is purely experimental; its structure and organization may change significantly at any time. There is no explicit support for parallel computing in the Extensible PDE Solvers package.

Chapter 1

Introduction

Wouldn't it be nice to be able to write a 25-line code that could efficiently set up and solve a class of PDEs on an arbitrary geometry (see Fig. 1.1)? The Extensible PDE Solvers package is a first attempt at providing such a programming environment.

```
main(int Argc, char **Args){
/* define the variables */
    DDGrid              *grid;
    DDPDEDiscretization *disc;
    DDFunction          *f, *g;
    DDDFunction         *u;
    DDPDE               *pde;
    DDBC                *bc;
    DDBCDiscretization  *bcdisc;
    DDOneGrid           *onegrid;
    DDDomain            *domain;
/* define the grid */
    grid = DDGridLoad(''structure.grid'');
/* define the PDE */
    f = DDFunctionCreate(2,2,rhs1,rhs2,&ectx);
    pde = DDPDECreateIsoLinearElasticity2(f,1.,.3,0,0);
/* define the discretization */
    disc = DDPDEDiscretizationCreate(DDPDEDISCFE2dLIN);
    DDPDEDiscretizationSetUp(disc);
/* define the Dirichlet boundary conditions */
    g = DDFunctionCreate(2,2,dir1,dir2,0);
    bc = DDBCCreateDirichlet(g,0,0);
/* define the discretization of boundary conditions */
    bcdisc = DDBCDiscretizationCreateSimple();
/* setup and solve discrete system */
    domain = DDDomainCreate(grid,disc,pde);
    DDDomainAddBoundary(domain,bc,bcdisc);
    onegrid = DDOneGridCreateWithCommandLine(domain,&Argc,Args);
    DDOneGridSetUpWithCommandLine(onegrid,&Argc,Args);
    u = DDOneGridSolve(onegrid);
}
```

Figure 1.1: Example from the extensible PDE solvers package

The Extensible PDE Solvers package provides a powerful, yet easy-to-use, interface to numerical methods for elliptic PDEs. In addition, it is relatively easy to extend the set of methods.

The present features of the Extensible PDE Solvers package include

- a high-level, abstract interface to fundamental objects such as grids, discretizations, and boundary conditions;
- identical support for structured, semistructured, and unstructured grids;
- code reuse between finite element and finite difference discretizations; and
- the ability to extend the code to new discretizations, PDEs, and linear system solvers without modifying a single line of the Extensible PDE Solvers package.

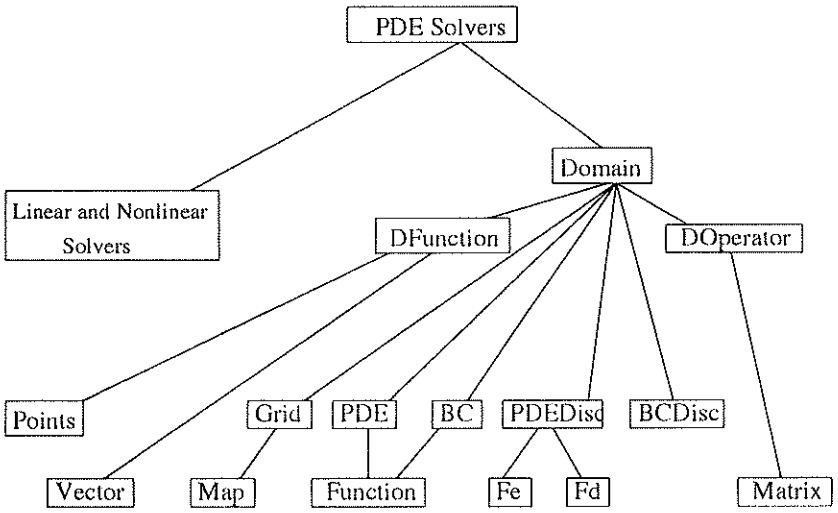

Figure 1.2: Outline of fundamental objects

The goals of the Extensible PDE Solvers package (and, more generally, for all of the Portable, Extensible Toolkit for Scientific computation (PETSc) [3], developed by Gropp, McInnes, and Smith) are

- flexibility,
- reusability,
- good efficiency for both model and "real-world" problems, and
- the ability to allow a programmer to set up and solve a problem quickly.

To achieve these goals, the Extensible PDE Solvers package was designed using the concept of data encapsulation. Simply put, only routines that absolutely must know the data storage format of an object are allowed to directly manipulate that data. All other routines can access the data only by calling these special (data access) routines. Different software components communicate with each other through a small, well-defined set of interfaces. (In our current implementation, this concept is occasionally violated in the interest of getting things done quickly.) See Figure 1.2 for an overview of some of the components in the Extensible PDE Solvers package.

1.1 Accessing the Guts

Despite the apparent power and flexibility of the Extensible PDE Solvers package, this package actually provides a simplified interface to the more flexible and powerful routines that are part of the PETSc package.

The Extensible PDE Solvers package is designed to sit on top of these more powerful, but more complicated, routines and make it easier for you to solve PDEs without having to spend a large amount of time coding. You do not need to know about or understand the lower-level routines in order to use the package efficiently. If you find that the Extensible PDE Solvers package does not give you the functionality that you need, you should then (and only then) investigate these other parts of PETSc.

1.2 Why C?

The Extensible PDE Solvers package (and most of PETSc) is coded in C. It is intended to be usable from Fortran 77 and (in the future) C++, but C is used as the basic library language for the following reasons:

- maturity of language and compilers,
- ability to use data encapsulation techniques,
- portability across virtually all machines, and
- ability to be called easily from either Fortran 77 or C++.

The Fortran 77 interface to the Extensible PDE Solvers package has not been extensively tested. We do not recommend using Fortran 77 with the Extensible PDE Solvers package. The Fortran 77 interface for the rest of PETSc is, however, well tested and used by many people.

1.3 Further Information

This manual mentions some of the routines in the Extensible PDE Solvers package; however, usage instructions are provided only for the more common routines. More detailed information about the routines mentioned in this manual may be found in Chapter 7 and the man pages using `toolman`, one of the utilities provided by PETSc for accessing the detailed documentation on the routines.

This package is growing through the addition of new routines. Suggestions (and bug reports) should be e-mailed to `bsmith@mcs.anl.gov`.

The Extensible PDE Solvers package includes some graphical aids for displaying, for example, the progress of the solution algorithm. It is possible to install the Extensible PDE Solvers package without the X-Windows graphics, but we do not recommend it.

1.4 Installing the Package

The Extensible PDE Solvers Package is available, as part of PETSc, by anonymous ftp from `info.mcs.anl.gov` in the directory `pub/pdetools`. The `readme` file indicates which compressed tar files you should obtain. Once the tar file has been extracted at your site, the `readme` file contains instructions on the compiling and installation of the software.

To make the examples for the Extensible PDE Solvers package, change to the directory `domain/tools`, and type `make BOPT=g ex1`. This will make the first example. See Chapter 5, where all of the examples are discussed.

1.5 Error Messages

The debugging version of the PETSc package will generate error tracebacks of the form

`Line` *linenumber* `in` *filename: message*
`Line` *linenumber* `in` *filename: message*
...
`Line` *linenumber* `in` *filename: message*

if an error is detected. The first line indicates the file where the error was detected; the subsequent lines give a traceback of the routines that were calling the routine that detected the error. A message may or may not be present; if present, it gives more details about the cause of the error.

The production libraries are often built without the ability to generate these tracebacks (or even detect many errors). If your programs crash unexpectedly, try to compile the debugging version and run that.

Chapter 2

Working with Grids

The Extensible PDE Solvers package has support for many types of grids. Implementations are provided both for tensor product grids with an optional mapping and for the usual unstructured grids.

All of the grids are organized around a variable type called a `DDGrid`. These grids can be created, used for various purposes, and then destroyed. For instance, the following example displays a grid on your monitor.

```
DDGrid *grid;
grid = DDGrid2dCreateUniform(nx,ny,0.0,1.0,0.0,1.0);
DDGridDraw(window,0,grid,0);
DDGridDestroy(grid);
```

The basic, predefined grid operations are as follows:

- refine the grid,
- coarsen the grid,
- draw the grid,
- draw the boundary of the grid,
- partition the grid,
- copy the grid,
- determine whether points are in the grid,
- determine whether points are on the boundary of the grid,
- save the grid to a file, and
- read the grid from a file.

2.1 Tensor Product Grids

The simplest grids are tensor product grids in two or three dimensions. They may either be uniformly spaced grids or have nonuniform spacing between grid points. To create a uniformly spaced grid, use the command

```
grid = DDGrid2dCreateUniform(nx,ny,xmin,xmax,ymin,ymax);
```

or

```
grid = DDGrid3dCreateUniform(nx,ny,nz,xmin,xmax,ymin,ymax,zmin,zmax);
```

The arguments `nx,ny,nz` indicate the number of grids points, including the endpoints, while `xmin,xmax`, etc. indicate the extreme corners of the grid.

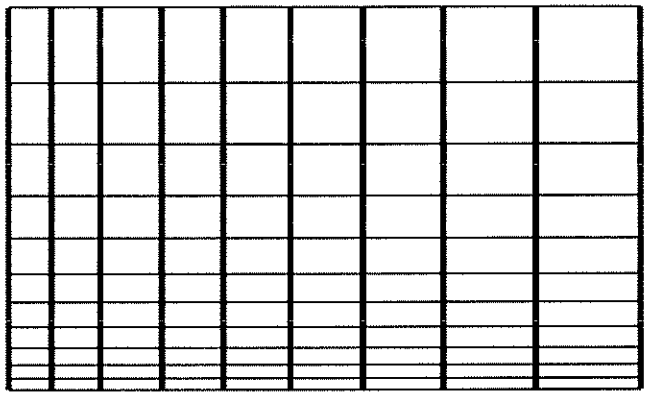

Figure 2.1: Tensor product grid

Nonuniformly spaced tensor product grids (see Figure 2.1) are created with either the command

```
grid = DDGrid2dCreateTensor(nx,ny,x,y);
```

or, in three dimensions,

```
grid = DDGrid3dCreateTensor(nx,ny,nz,x,y,z);
```

The arguments `x,y`, and `z` are double precision arrays containing the locations of the grid points along the various axes.

Adding a Mapping

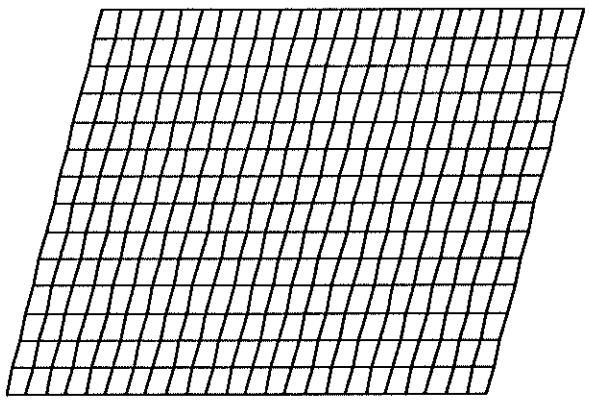

Figure 2.2: Mapped tensor product grid

You may associate with a uniform or tensor product grid a mapping to define regions that are not rectangular but merely logically rectangular (see Figure 2.2). This is done in two steps: first, define the mapping function, then, associate that mapping function with the grid.

To define a mapping function, you must provide two C routines that evaluate the mathematical function at a point and at a set of points (see below for the definition of points). For instance,

```
void MapPointAffine2d(inpoint,outpoint,p)
DDPoint   *inpoint, *outpoint;
void      *p;
{
  outpoint->x = 3.0*inpoint->x + 2.0*inpoint->y + 0.0;
  outpoint->y = 2.0*inpoint->x + 1.0*inpoint->y + 0.5;
}
```

is the code for a C function that evaluates the function

$$\begin{pmatrix} x_{out} \\ y_{out} \end{pmatrix} = \begin{pmatrix} 3.0 & 2.0 \\ 2.0 & 1.0 \end{pmatrix} \begin{pmatrix} x_{in} \\ y_{in} \end{pmatrix} + \begin{pmatrix} 0.0 \\ .5 \end{pmatrix}.$$

A corresponding C routine for a set of points is given by

```
void MapMeshAffine2d(mesh,points,p)
DDMesh    *mesh;
DDPoints  *points;
void      *p;
{
  int       nx = mesh->nx, ny = mesh->ny, i, j, ii;
  double    *xout = points->x, *yout = points->y;
  double    c, d, *x = mesh->x, *y = mesh->y;
  for ( i=0; i<ny; i++ ) {
    c = 2.0*y[i]; d = y[i] + .5;
    ii = i*nx;
    for ( j=0; j<nx; j++ ) {
      xout[ii+j] = c; yout[ii+j] = d;
    }
  }
  for ( i=0; i<nx; i++ ) {
    c = 3.0*x[i]; d = 2.0*x[i];
    for ( j=0; j<ny; j++ ) {
      xout[i+j*nx] += c; yout[i+j*nx] += d;
    }
  }
}
```

Once the two C functions have been written, a map is defined by

```
DDMap *map;
map  = DDMap2dCreate(MapPointAffine2d,MapMeshAffine2d,(void *)0);
```

The final argument to `DDMap2dCreate()` is an optional *function context*, it contains a pointer to any additional data needed by the C routines. This is always passed to the C routines through the final `void *p` argument (this is unused in the above example.) We give an example of its use in Section 2.5.

Now that a map has been created, it is added to a grid using the command

```
DDGridAddMap(grid,map);
```

2.2 Unstructured Grids

The Extensible PDE Solvers package also has support for the common unstructured grids.

2.2.1 Grids Based on Triangles

Perhaps the simplest unstructured grid in two dimensions is based on triangles. These may be created in several ways. The easiest is by converting from a uniform, tensor product or mapped grid (or quadrilateral grid that are introduced below) using the command

```
DDGridToTriangles(grid);
```

The second way to generate unstructured grids based on triangles is from a list of nodes and triangles, using the command

```
grid = DDGridCreateTriangles(nv,x,y,nt,nodes,nb,bnd);
```

The first argument, `nv`, is the total number of vertices. The number of triangles is given by `nt`. The coordinates of the nodes are stored in the double precision arrays `x` and `y`. The integer array `nodes` of dimension `3*nt` contains the indices of the vertices of each triangle. The indices start at 0, not 1.

In addition, it is convenient to keep information about the boundary of the grid. This is done by providing a list of line segments that define the boundary. The integer `nb` is two times the number of line segments on the boundary, while the integer array `bnd` contains the vertices of the nodes that define these line segments. For each line segment the interior of the domain must be to the left of the line segment.

In a simple example consider the unit square divided into two triangles. The input information for `DDGridCreateTriangles` could be given by

```
nv = 4;
x[0] = 0.0;  y[0] = 0.0;  x[1] = 1.0;  y[1] = 0.0;
x[2] = 0.0;  y[2] = 1.0;  x[3] = 1.0;  y[3] = 1.0;
nt = 2;
nodes[0] = 0; nodes[1] = 1; nodes[2] = 2;
nodes[3] = 3; nodes[4] = 2; nodes[5] = 1;
bn = 4;
bnd[0] = 0; bnd[1] = 1;  bnd[2] = 1; bnd[3] = 3;
bnd[4] = 3; bnd[5] = 2;  bnd[6] = 2; bnd[7] = 0;
```

The third technique for creating triangular grids in the Extensible PDE Solvers package is by providing boundary information (and optional interior vertex information) and having the software automatically generate the triangulation. This triangulation is generated using software provided by Timothy Baker of Princeton University (the code is in `domain/grid/bapet.f`). To generate the grid use the command

```
grid = DDTriangulateBaker(x,y,bnd,nb,ni,nbnd);
```

The integers `nb` and `ni` contain the number of vertices on the boundary and in the interior. The double precision arrays `x` and `y` contain the locations of the vertices with those on the boundary listed first. The integer `nbnd` contains the number of line segments that define the boundary, while `bnd` is an integer array containing the indices of the end points of the line segments. For each line segment the interior of the domain must be to the left of the line segment.

It is also possible to input the boundary information (and optional interior points) using X-Windows graphics and the mouse. This is done with the commands

```
grid = DDGridInputTriangularGrid(win);
```

The argument `win` is the name of an X-Window. These windows are introduced below.

2.2.2 Grids Based on Quadrilaterals

In addition to grids based on triangles, the other standard unstructured grid in two dimensions provided with the Extensible PDE Solvers package is a collection of quadrilaterals. The easiest way to create such a grid is by converting from a uniform, tensor product or mapped grid using the command

```
DDGridToQuadrilaterals(grid);
```

The second way to generate unstructured grids based on quadrilaterals is from a list of nodes and quadrilaterals using the command

```
grid = DDGridCreateQuadrilaterals(nv,x,y,nq,nodes,nb,bnd);
```

The first argument, `nv`, is the total number of vertices. The number of elements is given by `nq`. The coordinates of the nodes are stored in the double precision arrays `x` and `y`. The integer array `nodes` of dimension `4*nq` contains the indices of the vertices of each element. The boundary information is passed in using the same format as that for triangle-based grids.

2.2.3 Grids Based on Tetrahedrons

One way to construct an unstructured mesh in three dimensions is using tetrahedrons. A tetrahedral grid can be generated in one of two ways: from a uniform, tensor product or mapped grid (or from a hexahedral grid introduced below) or from a list of tetrahedrons. To generate a tetrahedral grid from a uniform, tensor product or mapped grid (or grid of hexahedrons), use the command.

```
DDGridToTetrahedrals(grid);
```

To generate a tetrahedral grid from a list, use the command

```
grid = DDGridCreateTetrahedrals(nv,x,y,z,nt,vert,bn,bound);
```

The arguments `nv` and `nt` contain the number of grid vertices and tetrahedrons, respectively. The double precision arrays `x,y`, and `z` contain the locations of the vertices, while `vert` is an integer array of dimension `4*nt` that contains the node numbers of the vertices of the tetrahedrons. The nodes are ordered in the following way: first the front lower left, then the front lower right, then the top node followed by the node in the back. The integer `bn` contains the number of triangles that define the boundary, and `bound` is an integer array of size `3*bn` containing the indices of the vertices of the boundary triangles. Using the right-hand rule, the normals to the boundary triangles point out of the domain.

2.2.4 Grids Based on Hexahedrons (Bricks)

Another way to construct an unstructured mesh in three dimensions is by using hexahedrons. A hexahedral grid can be generated in one of two ways: from a uniform, tensor product or mapped grid or from a list of hexahedrons. To generate a hexahedral grid from a uniform, tensor product or mapped grid, use the command

```
DDGridToHexahedrals(grid);
```

To generate a hexahedral grid from a list, use the command,

```
grid = DDGridCreateHexahedrals(nv,x,y,z,nt,vert,bn,bound);
```

The arguments `nv` and `nt` contain the number of grid vertices and hexahedrons, respectively. The double precision arrays `x`,`y`, and `z` contain the locations of the vertices, while `vert` is an integer array of dimension `8*nt` that contains the node numbers of the vertices of the hexahedrons. The nodes are ordered in the following way: first the front lower left, then the front lower right, then the top right, the top left, the back lower left, the back lower right, the back upper right, and the back upper left. The integer `bn` contains the number of quadrilaterals that define the boundary, and `bound` is an integer array of size `4*bn` that contains the indices of the vertices of the boundary quadrilaterals. Using the right-hand rule, the normals to these quadrilaterals point out of the domain.

2.2.5 Adding a Mapping

Though conceptually it is possible to think of mapping an entire unstructured grid, we deem this unnecessary, since unstructured grids by their nature can be constructed for complicated geometries. However, it is desirable for unstructured grids to support curved boundaries. This capability is obtained by providing an optional mapping function that is applied to only the boundary nodes during grid refinement.

The following example defines a mapping that creates a Pacman-type domain:

```
#define SIGN(a) ((a) < 0 ? -1.0 : 1.0)
void MapPointCircle(inpoint,outpoint,circle)
DDPoint   *inpoint, *outpoint;
CircleCtx *circle;
{
  double x = inpoint->x - circle->x;
  double y = inpoint->y - circle->y, theta;
  if (x == 0.0) y = SIGN(y)*circle->r;
  else {
    theta = atan((y/x)*SIGN(x)*SIGN(y));
    if (theta > circle->theta+.0000001 || x < 0.0) {
      x = SIGN(x)*circle->r*cos(theta);
      y = SIGN(y)*circle->r*sin(theta);
    }
  }
  outpoint->x = circle->x + x; outpoint->y = circle->y + y;
}
```

This mapping is then assigned to a grid, using the following code:

```
map  = DDMap2dCreate(MapPointCircle,(void *)0,&circle);
DDGridAddBoundaryMap(grid,map);
```

In the previous example the function maps only a single point. A similar C function could certainly be written to map a set of points.

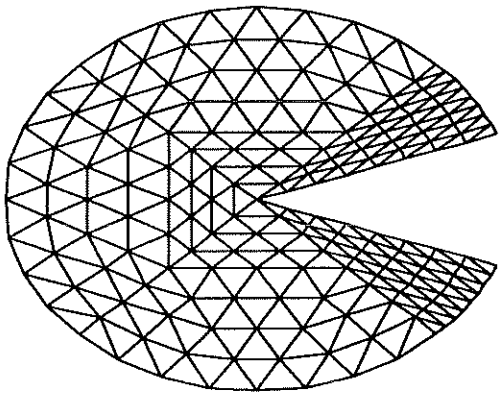

Figure 2.3: Pacman grid from a mapping

Once the grid has been refined several times, we obtain the grid as depicted in Figure 2.3. Note that this grid is probably not a good grid for the given domain; it is intended just as an example of how mapped grids may be created.

2.3 Operations on Grids

There are a variety of pre-defined operations you may apply to grids.

2.3.1 Refining Grids

All of the grids may be refined by using the command

```
DDGridRefine(grid);
```

Obviously, refinement means something slightly different for each different grid.

The uniform, tensor product and mapped grids are refined by inserting a new set of mesh lines halfway between each mesh line on the original grid. The triangular grids are refined by dividing each element into four similar elements by bisecting the edge of each side. Quadrilateral grids are refined by dividing into four elements. Hexahedral elements are divided into eight elements by inserting a new node at the center of each hexahedron, the center of each face, and the center of each edge. The tetrahedral elements are refined by inserting a new node at the center of each edge. For all the elements except the tetrahedron, the refinement ensures shape regular elements.

It is desirable to be able to perform partial refinement of a grid. However the Extensible PDE Solvers package currently contains no code to do this, since it is somewhat tricky.

2.3.2 Grid Coarsening

It is sometimes useful to coarsen a grid automatically, for instance, if one wishes to apply multigrid when only the finest grid is given. Grid coarsening may be done using the Extensible PDE Solvers package by using the command

```
DDGridUnRefine(grid,&newgrid);
```

Since unrefining can be a complicated matter, it is possible to modify the way a grid is coarsened. with the command

```
DDGridAddUnRefineContext(grid,...);
```

See the manual pages for the latest version of this command.

At the moment the only type of unstructured grid that may be coarsened is a grid of triangles. The package does this by first computing a maximal independent set of the boundary and interior nodes of the grid and then triangulating these using the code of T. Baker. The structured grids are coarsened by removing every second grid line in all directions.

2.3.3 Saving and Loading Grids

It is possible to save any grid to a file (except the mappings) and read it back in at a later time. Saving a grid is done with the command

```
DDGridStore(grid,filename);
```

A grid may be read in from a file with the command

```
grid = DDGridLoad(filename);
```

2.3.4 Graphics

Before any graphic operations can be performed, at least one graphics window must be opened. To do so, use the commands

```
XBWindow win;
win  = XBWinCreate();
XBQuickWindow(win,hostname,WindowTitle,x,y,nx,ny);
```

The arguments are `win`, the window to open; `hostname`, the name of the display to open the window on (usually "" to get the default monitor); `WindowTitle`, the title to appear at the top of the window; `x,y` the location of the upper left corner of the window in the display ((0,0) denotes the upper left corner); and `nx,ny` the width and height of the window in pixels. For example,

```
win  = XBWinCreate();
if (XBQuickWindow(win,"","Grid",0,0,600,600)) {
  XBError(); exit(1);
}
```

These routines are all part of the `xtools` component of PETSc.

It is useful to be able to define the size and location of the window in grid coordinates (the same coordinates by which the grids are defined). This is done by using the `XBInfo` structure. For instance, to define a region that allows a window to view from `x = -1` to `x = 1` and `y = 0` to `y = 1`, use

```
XBInfo region;
region.xmin = -1.0; region.xmax = 1.0;
region.ymin =  0.0; region.ymax = 1.0;
region.hold = 1;
```

The variable `hold` contains an integer indicating how long in seconds the image should be held in the window before the program continues. `hold = 0` indicates that it should hold for no time, while `hold = -1` indicates it should hold until the user inputs a keystroke or mouse click. For two-dimensional images you should hit the carriage return, while for three-dimensional images the left mouse button allows you to rotate the image, and the center button indicates that the program should continue.

To display a grid's boundary, use the command

```
DDGrid   *grid;
XBWindow win;
XBInfo   region;
int      color;
DDGridDrawBoundary(win,&region,grid,color);
```

The arguments are the window in which to draw, the region that relates the grid coordinates to the window coordinates, the grid, and the color to draw the grid boundary. One may pass in `0` instead of `&region` to use the default region.

Displaying the entire grid is done with the command

```
DDGridDraw(win,&region,grid,color);
```

The grid data structure also has its own (secret) copy of a region. It uses this by default if the user does not pass in a region. To set the `hold` for a grid, use the command

```
DDGridSetHold(grid,holdvalue);
```

Here `holdvalue` indicates the time in seconds to hold the figure.

It is also possible for two-dimensional grids to allow the user to "zoom" in on a portion of the grid. The command

```
int      color;
XBWindow win;
DDGrid   *grid;
DDGridZoom(win,0,grid,color,0);
```

displays the grid. The user may "zoom" in by pressing the left mouse button and "zoom" out by pressing the center mouse button. The right mouse button returns from the function.

2.3.5 Partitioning Grids

It is sometimes useful to partition a grid into several (possibly overlapping) subgrids. This partitioning is, for example, useful for block iterative methods (such as overlapping Schwarz). Partitioning is done with the command

```
DDGrid     *grid;
int        minsize,levels,overlap;
IndexArray *indices;
indices = DDGridPartition(grid,minsize,levels,overlap);
```

The argument `minsize` is the minimum number of nodes allowed in a subgrid, while `overlap` is the number of levels of overlap between subgrids. The number of subgrids is given by `2^overlap`. The structure `IndexArray`, which is a linked list of node numbers in each subgrid, is given by

```
struct _IndexArray {
  int                n, *ii;
  struct _IndexArray *next;
};
typedef struct _IndexArray IndexArray;
```

The integer array `ii` contains the indices for the subdomains, and `n` is the number of nodes in the list.

Since there are many ways to partition a grid, the Extensible PDE Solvers package comes with several defaults. The first is to use a crude recursive spectral bisection method to do the partitioning. This is effective for small grids, but is very slow for grids with more than a few hundred unknowns. You can also use the natural ordering of the nodes in the grid to do the partitioning. This generally gives bad partitions but does have the advantage of being very fast to calculate. A grid can be forced to use the naive partitioner with either the command

```
DDGridUseNaivePartitioner(grid);
```

or

```
DDGridUseNaivePartitionerWithCommandLine(grid,argc,args);
```

If the string `-naive` is in the command line, then the naive partitioner is associated with the given grid.

The Extensible PDE Solvers pakcage also has interfaces to several other partitioning packages. These can be accessed with the commands

```
DDGridUsePamPartitionerWithCommandLine(grid,argc,args);
DDGridUseBaSiPartitionerWithCommandLine(grid,argc,args);
DDGridUseChacoPartitionerWithCommandLine(grid,argc,args);
```

The Pam partitioner was written by Françoise Lamour and Patrick Ciarlet, the BaSi partitioner was written by Stephen Barnard and Horst Simon, and the Chaco partitioner was written by Bruce Hendrickson and Robert Leland. Refer to the software to determine how these packages may be obtained. The Pam partitioner is a fast, greedy partitioner, while the other two are slower methods based on recursive spectral bisection.

2.4 Manipulating Points

It is sometimes useful to be able to manipulate points in space. The Extensible PDE Solvers package has the concept of both a point, `DDPoint` and a set of points, `DDPoints`. A set of points is created with the commands

```
DDPoints *pts;
pts = DDPoints2dCreate(n);
pts = DDPoints3dCreate(n);
```

where `n` is the number of points needed.

Various basic operations can be performed on points, for instance,

```
pts2 = DDPointsCopy(pts1);
pts3 = DDPointsUnion(pts1,pts2);
pts3 = DDPointsDifference(pts1,pts2);
DDFunction *function;
pts2 = DDPointsWithNonZeroFunction(pts1,function);
```

The final routine DDPointsWithNonZeroFunction() returns all the points for which the given DDFunction is nonzero. See below for the definition of DDFunction.

It is also possible to draw points in a window with the command

```
DDPointsDraw(win, &region, pts, color);
```

It is possible to determine which points lie in a grid with the command

```
pts2 =  DDGridGetPointsIn(grid,pts1,on);
```

The integer flag on should be 1 if points on the boundary of the grid are to be included; otherwise it should be 0. Points on the boundary of a domain can be found by using

```
pts2 = DDGridGetPointsOnBoundary(grid,pts1);
```

2.5 Manipulating Mathematical Functions

In defining PDEs we must be able to represent mathematical functions in a convenient way. This is done in the Extensible PDE Solvers package using a DDFunction. A DDFunction is simply a realization of a mathematical function. A function is created with the command

```
DDFunction *function;
int        in, out;
void       (*f1)(),(*f2)(),*context;
function = DDFunctionCreate(in,out,f1,f2,context);
```

The mathematical function has in input variables and is (possibly) vector valued with out components. The argument (*f1)() is a user-provided C function that takes three arguments: a DDPoint *, a double *, and the pointer to a function context, context. The argument (*f2)() is a similar C function whose first argument is DDPoints *. The function context, context, is a way of getting any needed information into the user's C function. For example, if the mathematical function we wish to implement has two parameters and is given by

$$f(x_1, x_2) = \alpha x_1 + \beta x_2,$$

then the C functions could be written as follows.

```
typedef struct {double alpha, beta;} MyContext;
void f1(pt,out,ctx);
double    *out;
DDPoint   *pt;
MyContext *ctx;
{
  *out = ctx->alpha*pt->x + ctx->beta*pt->y;
}
```

```
void f2(pts,out,ctx);
double    *out;
DDPoints  *pts;
MyContext *ctx;
{
  int i, n = pts->n;
  for ( i=0; i<n; i++ ) {
    out[i] = ctx->alpha*pts->x[i] + ctx->beta*pts->y[i];
  }
}
```

A DDFunction may be evaluated at a set of points by using the command

```
DDDFunction *u;
DDPoints    *pts;
u = DDFunctionEvaluatePoints( func, pts, 0 );
```

This routine returns a discrete function containing the values of the function at the given points. A DDDFunction contains essentially an array of doubles with the values of the function and a copy of the points at which it was evaluated.

2.6 Adding New Types of Grids

It is possible to add new types of grids to the package or to modify those already there. You must provide functions that perform the grid operations. Begin by looking at the file domain/grid.h. and then the simplest grid implementation that is contained in domain/grid/mesh2d.c.

Chapter 3

Working with PDEs and Discretizations

This chapter explains how you define the partial differential equation and the discretization to be used to find approximate solutions.

3.1 Defining the PDE

The Extensible PDE Solvers package comes with two default PDEs: the scalar convection-diffusion equation (actually, in the present version, the convection is ignored), and the equations of isotropic linear elasticity. Other simple, second order linear elliptic equations could be added. To solve other problems, the biharmonic for example, would require some reorganization.

To define a convection-diffusion PDE in two dimensions of the form

$$-\nabla \begin{pmatrix} ax(x,y) & 0 \\ 0 & ay(x,y) \end{pmatrix} \nabla u = f,$$

you must first define the functions `ax()`, `ay()`, and `f()` and then create the PDE. You may be do this by using the commands

```
DDFunction *f,*ax,*ay;
DDPDE      *pde;
f = DDFunctionCreate(2,1,rhs1,rhs2,0);
ax = DDFunctionCreate(2,1,ax1,ax2,0);
ay = DDFunctionCreate(2,1,ay1,ay2,0);
pde = DDPDECreateConvectionDiffusion2(f,ax,ay,0,0,0);
```

The final three arguments (where zero is passed above) are for the convection terms and the Helmholtz term. At the moment they are ignored. If you would like to use default values of `-1` for `ax` and `ay`, simply pass in `0` as the argument.

```
pde = DDPDECreateConvectionDiffusion2(f,0,0,0,0,0);
```

The convection-diffusion equation in three dimensions can be created with

```
pde = DDPDECreateConvectionDiffusion3(f,ax,ay,az,0,0,0,0);
```

Again, a zero can be passed in the locations of `ax`, `ay`, or `az` to represent the default function of `-1`.

You create a constant coefficient PDE for the equations of isotropic linear elasticity in two dimensions with the command

```
double E = 1.0, nu = .3;
pde = DDPDECreateIsoLinearElasticity2(f,E,nu,0,0,alpha);
```

For variable coefficients, use

```
DDFunction *E, *nu;
pde = DDPDECreateIsoLinearElasticity2(f,0,0,E,nu,alpha);
```

In three dimensions the only difference is that the `DDFunctions` `E` and `nu` are functions of 3 variables and the command is

```
pde = DDPDECreateIsoLinearElasticity3(f,0,0,E,nu);
```

The model used in two dimensions is of plane strain when `alpha=0` and plane stress when `alpha=1`.

Other PDEs may be defined by using the source code for one of those introduced above as a template.

3.2 Defining Boundary Conditions

The Extensible PDE Solvers package currently provides support for general Dirichlet and homogeneous Neumann boundary conditions. You may apply a combination of different boundary conditions on different parts of the boundary. To apply Dirichlet boundary conditions on the entire boundary, use the command

```
DDBC       *bc;
DDFunction *g;
bc = DDBCCreateDirichlet(g,(void *)0,(void *)0);
```

On the piece of the boundary defined by a characteristic function, `charf`, use

```
DDBC       *bc;
DDFunction *g, *charf;
bc = DDBCCreateDirichlet(g,charf,0);
```

On the piece of the boundary that lives in another grid, use

```
DDBC       *bc;
DDFunction *g;
DDGrid     *anothergrid;
bc = DDBCCreateDirichlet(g,0,anothergrid);
```

For homogeneous Neumann boundary conditions, use

```
bc = DDBCCreateNeumann(0,0,0,0,charf,anothergrid);
```

The first four arguments are, at present, ignored. None, or only one, of the last two arguments should be given.

3.3 Using a Discretization

Creating a discretization simply requires calling the routine

```
DDPDEDiscretization *disc;
disc = DDPDEDiscretizationCreate(type);
DDPDEDiscretizationSetUp(disc);
```

The `type` can be any of `DDPDEDISCFD5pt`, `DDPDEDISCFD7pt`, `DDPDEDISCFE2dLIN`, `DDPDEDISCFE3dTRILIN`, `DDPDEDISCFE2dBILIN`, or `DDPDEDISCFE3dLIN`.

For some of the finite element discretizations, it is possible to set the particular numerical integration scheme that is to be used. This is done with

```
DDPDEDiscretizationSetIntegrationScheme(disc,order,type);
```

The integer arguments `order` and `type` indicate the order of the numerical integration scheme you would like used and the particular scheme of that order. Usually `type` is set to zero. This routine should be called after `DDPDEDiscretizationCreate` and before `DDPDEDiscretizationSetUp`.

Implementing the Discretizations

This section may be skipped by the causal reader who is mainly interested in using the package and less interested in its design.

The implementation for finite differences is straightforward, achieves little code reuse, and is essentially not data structure neutral. The implementation was just enough to work for the diffusion term in the convection diffusion equation.

The implementation for finite elements is more ambitious and tries to obtain code reuse between different PDEs and different types of elements. For instance, most of the element stiffness code is shared by all the elements. It is also designed in a data-structure-neutral way to facilitate the addition of new elements or PDEs. The basic design is as follows:

- `DDPDEDiscretetizationSetUp()` determines the numerical integration points on the reference element and evaluates the shape functions and their derivatives at the nodes.
- `DDDomainDiscretize` loops over all the elements and calls
- `DDiUniversalStiffnessElement` on each element, which loops over the integration points and calculates the derivatives of the shape function by using the Jacobian of the mapping.
- The contribution to the stiffness matrix for a single integration point is calculated by calling the bilinear form defined for the particular type of PDE being solved.

3.4 Discretizing the Boundary

The Extensible PDE Solvers package is remarkably simple minded about discretizing the boundary conditions. At the moment you should use the command

```
DDBCDiscretization *bcdisc;
bcdisc = DDBCDiscretizationCreateSimple();
```

Chapter 4

Solving a PDE

Solving a PDE requires the following steps:

- define the grid,
- define the PDE,
- define the discretization of the PDE,
- define the boundary condition(s), and
- define the discretization of the boundary conditions.

Once these tasks have been performed, all of the information is gathered together into a variable of type `DDDomain` with the commands

```
domain = DDDomainCreate(grid,disc,pde);
DDDomainAddBoundary(bc,bcdisc);
```

You may call `DDDomainAddBoundary` several times with different boundary conditions for different parts of the boundary.

Now we are ready to discretize and solve the PDE. The universal interface for this is the `DDDomainSolver`. A complete linear, second-order elliptic PDE solver maybe written as

```
DDDomainSolver *ds;
ds = DDDomainSolverCreateWithCommandLine(domain,argc,argv);
DDDomainSolverSetUpWithCommandLine(ds,argc,argv);
DDDomainSolverSolve(ds);
```

This one set of commands gives access to all of the linear system solvers, including several variants of multigrid. The possible command line arguments are listed in the next two sections.

The solvers come with a set of interfaces that allow the approximate solution and error to be visualized during the solution process. This may be done in several ways. The simplest is to plot a line graph of the norm of the residual and the error at each iteration of the iterative method. This may be done with the commands

```
XBWindow       win;
DDDomainSolver *ds;
...
win = XBWinCreate();
XBQuickWindow(win,"","Residual",600,0,300,300);
DDDomainSolverAddLineGraph(ds,win);
```

If you wish to plot the actual solution or error, you may use the commands

```
XBWindow win1, win2;
...
DDDomainSolverAddWindow1(ds,win1,0);
DDDomainSolverAddWindow2(ds,win2,0);
```

The (optional) third argument is a `XBInfo` region that would define which portion of the window is to be used for displaying the solution (or error). In two dimensions the approximate solution and error are displayed using a color contour plot. In three dimensions there is at present no code for displaying the approximate solutions.

Of course, to visualize the error, the solver code must know the exact solution. You can tell the `DomainSolver` the exact solution by defining a `DDFunction` for that solution and then calling

```
DDFunction *solution;
DDDomainSolver *ds;
DDDomainSolverAddSolution(ds,solution);
```

The solvers may also symbolically display the sparse matrix representation of the discretized operator. In addition, if a direct LU factorization is used, it may display the reordered matrix used to decrease fill. Several of the examples demonstrate this capability.

4.1 Classical Solvers

The command `DDDomainSolverSolve` has access to all of the linear system solvers in the `Simplified Linear Equation Solvers (SLES)` component of PETSc [1]. It also has access to all of the Krylov space methods in `KSP (Krylov Space Package)` [2]. They may be accessed with command line options.

The command line options for the classical solvers are

```
-itmethod ITMETHOD - Krylov space method to use
-restart n - restart for GMRES
-svmethod SVMETHOD - type of preconditioner or direct method
-tol t -  tolerance on decrease in residual
-its n - maximum number of iterations
-fill n - levels of fill to use if ILU method is chosen
-hold n - time to display each window, -1 means wait for input
-subdomains m - number of subdomains for Schwarz methods
-overlap n - overlap to use for Schwarz methods
-additive - use additive Schwarz method, else defaults to multiplicative
-showgrid - draws grid on contour plots
-surface - draws surface plot rather than contour plots
-levels r - number of levels to refine grid
Subdomain options (if -svmethod osm used )
  -subsvmethod -subitmethod -subrestart -subtol -subits -subfill
Command Line Arguments for ITMETHOD
  richardson chebychev cg gmres tcqmr bcgs cgs tfqmr lsqr preonly cr
Command Line Arguments for SVMETHOD
  lu jacobi ssor ilu icc iccjp bdd osm nopre
```

It is also possible to write code that allows access only to the classical iterative methods and not the multigrid solvers; this may be done with the commands

```
DDOneGrid *ds;
ds = DDOneGridCreateWithCommandLine(domain,argc,argv);
DDOneGridSetUpWithCommandLine(ds,argc,argv);
DDOneGridSolve(ds);
```

4.2 Multigrid Solvers

In addition to the classical linear equation solvers, a variety of multigrid solvers may be used. To access the multigrid solvers, use the command line option `-multigrid`. The command line options for the multigrid solvers are

```
-itmethod method - Krylov space method to use
-restart n - restart for GMRES
-levels n - number of levels
-jacobi - use Jacobi smoothing
-gs - use Gauss-Seidel smoothing rather than default symetric GS
-cycles n - 1 for V cycle 2 for W cycle
-presmooths n - number of pre smoothing steps
-postsmooths n - number of post smoothing steps
-additive - use additive multigrid rather than traditional
-full - use full multigrid, not just cycles
-tol t -  tolerance on decrease in residual
-its n - maximum number of iterations
-hold n - time to display each window, -1 means wait for input
-surface - draws surface plot rather than contour plots
-showgrid - draws grid in contour plots
Command Line Arguments for ITMETHOD
  richardson chebychev cg gmres tcqmr bcgs cgs tfqmr lsqr preonly cr
```

It is also possible to compile code that has access to only the multigrid solvers and not the classical iterative methods. This may be done with the commands

```
DDMultiGrid *ds;
ds = DDMultiGridCreateWithCommandLine(domain,argc,argv);
DDMultiGridSetUpWithCommandLine(ds,argc,argv);
DDMultiGridSolve(ds);
```

In fact, the `DDDomainSolver` routines are simply wrappers that call either the `DDOneGrid` routines or the `DDMultiGrid` routines, based on command line arguments.

Chapter 5

Organization

5.1 Examples

The directory `domain/examples` contains over 50 complete examples that demonstrate different aspects of the Extensible PDE Solvers package. In addition, the examples directory contains a variety of two dimensional unstructured grids. Since the examples are undergoing constant revision, they may be slightly different from those indicated in this section.

- `ex1` - Indicates how different types of grids may be defined and drawn.
- `ex2` - Displays some simple grids in two dimensions.
- `ex3` - Gives contour plots of discrete functions in two dimensions.
- `ex4` - Saves a grid to a file.
- `ex5` - Loads grids from file and displays.
- `ex6` - Finds unions of sets of points and plots.
- `ex7` - Finds points in a domain.
- `ex8` - Displays uniform grid in three dimensions.
- `ex9` - Finds points in a three-dimensional domain and its boundary.
- `ex10` - Finds points in a three-dimensional mapped grid.
- `ex11` - Solves a Poisson problem on a uniform grid in two dimensions.
- `ex12` - Solves a Poisson problem on a uniform grid in two dimensions using multigrid.
- `ex13` - Solves a Poisson problem on a triangular grid.
- `ex14` - Solves a Poisson problem on a triangular grid using multigrid.
- `ex15` - Solves a Poisson problem on a mapped grid using multigrid.
- `ex16` - Solves a Poisson problem on a Pacman shaped domain using multigrid.
- `ex17` - Solves a Poisson problem on a annulus using a quadrilateral grid and multigrid.

- ex18 - Solves a Poisson problem on a uniform grid in three dimensions with the onegrid solver.
- ex19 - Solves a Poisson problem on a uniform grid in three dimensions with multigrid.
- ex20 - Solves a Poisson problem on a mapped grid in three dimensions with the onegrid solver.
- ex21 - Solves a Poisson problem on a mapped grid in three dimensions with multigrid.
- ex22 - Solves a Poisson problem on a sphere with the onegrid solver using a hexahedral grid.
- ex23 - Solves a Poisson problem on a sphere with multigrid using a hexahedral grid.
- ex24 - Solves a Poisson problem on a sphere with the onegrid solver using a tetrahedral grid.
- ex25 - Solves a Poisson problem on a sphere with multigrid using a tetrahedral grid.
- ex26 - Solves a variable Poisson problem on a uniform grid with onegrid solver.
- ex27 - Solves a variable Poisson problem on a mapped grid with onegrid solver.
- ex28 - Solves a variable Poisson problem on a uniform grid with onegrid solver in three dimensions.
- ex29 - Solves a variable Poisson problem on a mapped grid with onegrid solver in three dimensions.
- ex30 - Uses alternating Schwarz in two dimensions with a Poisson problem.
- ex31 - Uses alternating Schwarz in two dimensions with a Poisson problem.
- ex32 - Uses alternating Schwarz in two dimensions with a Poisson problem.
- ex33 - Uses alternating Schwarz in two dimensions with a Poisson problem.
- ex34 - Uses alternating Schwarz in three dimensions with a Poisson problem.
- ex35 - Solves a linear elasticity problem on a uniform grid in two dimensions.
- ex36 - Solves a linear elasticity problem on a uniform grid in two dimensions solved with multigrid.
- ex37 - Solves a linear elasticity problem on a uniform grid in three dimensions.
- ex38 - Solves a linear elasticity problem on a uniform grid in three dimensions with multigrid.
- ex39 - Uses alternating Schwarz in two dimensions with linear elasticity.
- ex40 - Uses alternating Schwarz in three dimensions with linear elasticity.
- ex41 - Solves a Poisson problem in two dimensions on a grid read from a file.
- ex42 - Solves a linear elasticity problem in two dimensions on a grid read from a file.
- ex43 - Solves a Poisson problem in two dimensions on a grid read from a file, using multigrid.
- ex44 - Solves a linear elasticity problem in two dimensions on a grid read from a file.

- ex45 - Reads a grid in from a file and displays it, allowing the user to zoom in parts of the grid.
- ex46 - Partitions grids from files.
- ex47 - Unrefines a grid read in from a file.
- ex48 - Triangulates a simple region drawn with the mouse.
- ex49 - Solves three dimensional Poisson problem on a grid read in from a file.
- ex50 - Solves three dimensional linear elasticity problem on a grid read in from a file.
- ex51 - Displays a three dimensional grid read in from a file.

5.2 Directories

The directories are organized in a logical fashion matching the abstract design.

```
bc.h       bcdisc.h comtype   dfunc.h     points.h
disc/      domain.c domain.h  dop.h       examples/
file/      func/    func.h    graphics/   graphics.h
grid.h     makefile map/      map.h       op/
part/      pde/     pde.h     pdedisc.h   points/
readme     schwarz/ solvers/  sparse/     xtools/
fdomain.h grid/
```

Each of the major abstract components grids, pdes, discretizations, boundary conditions, and boundary condition discretizations has its own include files. The include file domain.h is the master include file that should be included in all codes that use the Extensible PDE Solvers package. In addition, is a subdirectory for each component contains the source code for that component. The manual pages for each routine indicate the file and directory that contain the source for that routine.

Chapter 6

Future Possibilities

I may write a new version of the Extensible PDE Solvers package in the future, using and extending the techniques that we have developed over the past few years in writing PETSc. The new version would be developed using more of the ideas of ComponentWare. Each component (for instance grids) would be more independent, and the interfaces would be made cleaner, so that other objects cannot access the grid data structures at all. One of the drawbacks of the code as it is written is that certain objects have to check the type of other types of objects. For instance, the discretization code has hardwired checks on certain types of PDEs and grids. In ideal ComponentWare this type of coding would be unnecessary and, in fact, not even permitted. Whether it is possible to determine the most suitable abstractions to do this "correctly" is not completely clear to me.

Other features that it would be nice to add are

- support for staggered grids/mixed methods,
- support across parallel platforms,
- a TCL/TK interface, and
- other discretizations such as spectral, collocation, and Sinc methods.

If you find this package useful or interesting and have any ideas on how it may be better organized or presented, please send any comments to `bsmith@mcs.anl.gov`.

Chapter 7

Summary of Routines

This chapter contains a brief summary of the routines in the Extensible PDE Solvers package. An easier way to access the data is through the man pages using `toolman`.

Most of these routines require the include files

```
#include "tools.h"
#include "domain/domain.h"
#include "domain/solvers/solver.h"
#include "xtools/basex11.h"
```

Fortran 77 programs should include `domain/fdomain.h` and also the file `domain/solvers/fsolver.h`. A special note for Fortran programmers: all of the datatypes (such as DDGrid) must be declared as integers in the Fortran code.

void DDGridDestroy(grid) DDGrid *grid;	Destroys a grid created with DDGrid*Create*().
void DDGridRefine(grid) DDGrid *grid;	Refines a grid created with DDGrid*Create*().
void DDGridUnRefine(grid,newgrid) DDGrid *grid,**newgrid;	UnRefines a grid created with DDGrid*Create*(). Actually works only for triangular grids
DDGrid *DDGridCopy(grid) DDGrid *grid;	Copies grid
DDPoints *DDGridGetPointsIn(grid,points,bound) DDGrid *grid; DDPoints *points; int bound;	Returns the nodes on a grid.
DDPoints *DDGridGetPointsOnBoundary(grid,points) DDGrid *grid; DDPoints *points;	Returns the nodes on the boundary of a grid.
IndexArray *DDGridPartition(grid,size,levels,overlap) DDGrid *grid; int size, overlap, levels;	Partitions a grid into a bunch of smaller grids.
DDGrid *DDGridSubGrid(grid,index) DDGrid *grid; IndexArray *index;	Given a grid and a index array, returns the subgrid
DDGrid *DDGrid2dCreateUniform(nx,ny,xmin,xmax, ymin, ymax) int nx,ny; double xmin,xmax,ymin,ymax;	Creates a 2d uniform mesh. The mesh is logically rectangular; an optional mapping function may be added to map to the true coordinates using DDGridAddMap().
DDGrid *DDGrid2dCreateTensor(nx,ny,x,y) int nx,ny; double *x,*y;	Creates a 2d tensor product mesh. The mesh is logically rectangular; an optional mapping function may be added to map to the true coordinates using DDGridAddMap().
void DDGridAddMap(grid,map) DDGrid *grid; DDMap *map;	Adds a map to an already created grid. DDGrid must be of type UNIFORM or TENSOR.

DDGrid *DDGrid3dCreateUniform(nx,ny,nz,xmin,xmax, ymin,ymax,zmin,zmax) int nx,ny,nz; double xmin,xmax,ymin,ymax,zmin,zmax;	Creates a 3d uniform mesh. The mesh is logically rectangular; an optional mapping function may be added to map to the true coordinates using DDGridAddMap().
DDGrid *DDGrid3dCreateTensor(nx,ny,nz,x,y,z) int nx,ny,nz; double *x,*y,*z;	Creates a 3d tensor product mesh. The mesh is logically rectangular; an optional mapping function may be added to map to the true coordinates using DDGridAddMap()
DDGrid *DDGridCreateTriangles(numvert,x,y,numtri,vert, bn,bound) int numvert, numtri, *vert, bn, *bound; double *x, *y;	Creates a 2d unstructured grid using triangular elements.
void DDGridAddBoundaryMap(grid,map) DDGrid *grid; DDMap *map;	Allows a function to be given that defines the boundary of a grid. If the grid is refined, new boundary points are shifted so that they lie on the true boundary. Works only for unstructured grids.
DDGrid *DDGridCreateQuadrilaterals(numvert,x,y, numquad,vert,bn,bound) int numvert, numquad, *vert, bn, *bound; double *x, *y;	Creates a 2d unstructured grid using quadrilateral elements.
void DDGridToQuadrilaterals(grid) DDGrid *grid;	Takes a grid and converts to unstructured, using quadrilaterals.
void DDGridToTriangles(grid) DDGrid *grid;	Takes a grid and converts to unstructured, using triangles in two dimensions.
DDGrid *DDGridCreateHexahedrals(numvert,x,y,z, numbrick,vert,bn,bound) int numvert, numbrick, *vert, bn, *bound; double *x, *y, *z;	Creates a 3d unstructured grid using hexahedral (bricks are a special case with parallel sides) elements.
void DDGridToHexahedrals(grid) DDGrid *grid;	Takes a grid and converts to unstructured, using hexahedral elements.
DDGrid *DDGridCreateTetrahedrals(numvert,x,y,z, numtet,vert,bn,bound) int numvert, numtet, *vert, bn, *bound; double *x, *y, *z;	Creates a 3d unstructured grid using tetrahedral elements.
void DDGridToTetrahedrals(grid) DDGrid *grid;	Takes a grid and converts to unstructured, using tetrahedrons.
DDDFunction *DDDFunctionCreate(n,dimin,dimout) int n,dimin,dimout;	Builds a DDDFunction structure.
DDDFunction *DDDFunctionCopy(dfunc) DDDFunction *dfunc;	Copies a DDDFunction structure.
void DDDFunctionDestroy(f) DDDFunction *f;	Frees space created by DDDFunctionCreate()
void DDDFunctionAddPoints(f,p) DDDFunction *f; DDPoints *p;	Adds the points associated with a discrete function.
void DDDFunctionAddDomain(f,g) DDDFunction *f; DDDomain *g;	Adds a grid associated with a discrete function.
void DDDFunctionScatterInto(df,seg,n,v,l) DDDFunction *df; int seg,n,*l; double *v;	Scatters values from v into a particular segment of a discrete function
DDDFunction *DDDFunctionDisjointUnion(f1,f2) DDDFunction *f1, *f2;	Takes two DDDFunctions and forms their union assuming that they have no common elements.
DDDFunction *DDDFunctionEvaluatePoints(f, p, b) DDFunction *f; DDPoints *p; DDDFunction *b;	Evaluates a function at a set of points. Use DDFunctionEvaluatePoint() for a single point.
DDDFunction *DDFunctionEvaluateDomain(f, d, b) DDFunction *f; DDDomain *d; DDDFunction *b;	Evaluates a function at all points on domain. Use DDFunctionEvaluatePoint() for a single point
DDFunction *DDFunctionCreate(in,out,f1,f2,context) void (*f1)(),(*f2)(); int in,out; void *context;	Builds a DDFunction structure, from two C functions.
void DDDFunctionUpDateXBInfo(df,info,flag) DDDFunction *df; XBInfo *info; int flag;	Given a discrete function and an XBInfo, updates the XBInfo so that the discrete function will fit completely in the plotting frame
void DDDFunctionAbsoluteValue(df) DDDFunction *df;	Given a discrete function, replaces each component with its absolute value.
void DDDFunctionPrint(file,df) DDDFunction *df; FILE *file;	Prints the values in a discrete function

DDPoints *DDPoints2dCreate(n) int n;	Creates a DDPoints data structure.
DDPoints *DDPoints3dCreate(n) int n;	Creates a DDPoints data structure.
void DDPointsDestroy(points) DDPoints *points;	Destroys a DDPoints structure created with DDPoints2dCreate() or DDPoints3dCreate().
DDPoints *DDPointsCopy(points) DDPoints *points;	Copies a DDPoints structure created with DDPoints2dCreate() or DDPoints3dCreate()
DDPoints *DDPointsUnion(p1,p2) DDPoints *p1, *p2;	Given two sets of points, returns their union.
DDPoints *DDPointsDifference(p2,p1) DDPoints *p1, *p2;	Given two sets of points, returns all points in the first set that are not in the second.
DDPoints *DDPointsWithNonZeroFunction(allpoints,function) DDPoints *allpoints; DDFunction *function;	Returns the points for which the function was nonzero.
DDMap *DDMap2dCreate(point,mesh,ctx) void (*point)(),(*mesh)(), *ctx;	Builds a mapping structure.
DDMap *DDMap3dCreate(point,mesh,ctx) void (*point)(),(*mesh)(), *ctx;	Builds a mapping structure.
void DDMapMesh(mesh, points) DDMesh *mesh; DDPoints *points;	Maps a mesh of points. Use DDMapPoints() to map a set of points. Use DDMapPoint() to map a single point.
void DDGridToPS(grid,filename) DDGrid *grid; char *filename;	Prints two dimensional grid to a Postscript(TM) file.
void DDPointsDraw(window,region,points,color) DDPoints *points; XBWindow window; XBInfo *region; int color;	Draws a set of points.
void DDGridDraw(window,region,grid,color) DDGrid *grid; XBWindow window; XBInfo *region; int color;	Draws a grid.
void DDGridDrawZoom(window,inregion,grid,color,outregion) DDGrid *grid; XBWindow window; XBInfo *inregion,*outregion; int color;	Draws a grid; allows the user with a mouse to zoom in and out of the grid
void DDGridDrawBoundary(window,region,grid,color) DDGrid *grid; XBWindow window; XBInfo *region; int color;	Draws boundary of a grid.
void DDDrawDFunction(window,region,uin,type,wininfo) XBWindow window,wininfo; DDDFunction *uin; int type; XBInfo *region;	Draws a discrete function. Works only in two dimensions.
void DDDrawDFunctionZoom(window,inregion,uin,type, wininfo) XBWindow window,wininfo; DDDFunction *uin; int type; XBInfo *inregion;	Draws a discrete function. Works only in two dimensions.
void DDGridAddTriangleUnRefineContext(grid,type,win, region,trace) DDGrid *grid; int type,trace; XBWindow win; XBInfo *region;	Allows one to change the type of unrefinement algorithm to be used or display the unrefinement interactively.
DDGrid *DDGridInputTriangularGrid(win) XBWindow win;	Allows one to use the mouse to input a grid
DDPDEDiscretization *DDPDEDiscretizationCreate(type) DDPDEDISCTYPE type;	Builds a discretization structure.
void DDPDEDiscretizationRegister(name, sname, create) int name; char *sname; DDPDEDiscretization *(*create)();	Given a discretization name (DDDISCTYPE) and a function pointer, adds the discretization to the mesh package.
void DDPDEDiscretizationRegisterDestroy()	Frees the list of discretizations that have been registered by DDPDEDiscretizationRegister().
void DDPDEDiscretizationGetMethod(Argc, argv, remove, sname, method) int *Argc,remove; char *argv, *sname; DDPDEDISCTYPE *method;	Given the argument list, returns the selected discretization method.

int DDPDEDiscretizationHelp(argc,argv) int *argc; char **argv;	Given the argument list, prints a help message if -help is one of the arguments.
void DDPDEDiscretizationRegisterAll()	Registers all the discretizations in the mesh package. To prevent all the methods from being registered and thus save memory, copy this routine and register only those methods you desire.
DDBCDiscretization *DDBCDiscretizationCreateSimple()	Creates a BCDiscretization structure for finite elements or finite differences.
void DDPDEDiscretizationSetIntegra- tionScheme(disc,order,scheme) DDPDEDiscretization *disc; int order, scheme;	Sets the numerical integration scheme to be used with a particular finite element discretization. If the discretization is not finite elements, this call is ignored.
DDPDE *DDPDECreateIsoLinearElasticity2(f,E,nu,e,n,alpha) double E,nu; DDFunction *f,*e,*n; int alpha;	Creates a PDE structure for the Isotropic equations of elasticity.
DDPDE *DDPDECreateIsoLinearElasticity3(f,E,nu,e,n) double E,nu; DDFunction *f,*e,*n;	Creates a PDE structure for the equations of elasticity.
DDPDE *DDPDECreateConvectionDiffusion2(f,ax,ay,bx,by,c) DDFunction *f,*ax,*ay,*bx,*by,*c;	Creates a PDE structure for the convection diffusion equation.
DDPDE *DDPDECreateConvectionDiffusion3(f,ax,ay,az, bx,by,bz,c) DDFunction *f,*ax,*ay,*az,*bx,*by,*bz,*c;	Creates a PDE structure for the convection diffusion equation.
DDBC *DDBCCreateDirichlet(g,charf,grid) DDFunction *g,*charf; DDGrid *grid;	Creates a BC structure for Dirichlet boundary conditions. For the entire boundary, pass in 0 for the charf and 0 for the grid. For a piece of the boundary defined by another grid, pass in that grid (this is useful for doing alternating Schwarz).
DDDOperator *DDDOperatorCreate(m,n,nc,dim) int m,n,nc,dim;	Creates a holder for a discrete operator. A discrete operator is simply a (sparse) matrix plus information on the discrete function that it operates on.
void DDDOperatorDestroy(op) DDDOperator *op;	Destroys the holder for a discrete operator, also destroys the matrix in it.
void DDDOperatorApply(op,fin,fout) DDDOperator *op; DDDFunction *fin,*fout;	Applies a discrete operator to a discrete function
void DDDOperatorApplyTrans(op,fin,fout) DDDOperator *op; DDDFunction *fin,*fout;	Applies the transpose of a discrete operator to a discrete function
void DDDOperatorApplyAdd(op,fin,fout) DDDOperator *op; DDDFunction *fin,*fout;	Applies a discrete operator to a discrete function, and adds it to another discrete function
void DDDOperatorGaussSeidel(op,m,fin,fout) DDDOperator *op; DDDFunction *fin,*fout; int m;	Applies a sweep of Gauss-Seidel with a discrete linear operator
void DDDOperatorSymmetricGaussSeidel(op,m,fin,fout) DDDOperator *op; DDDFunction *fin,*fout; int m;	Applies a sweep of symmetric Gauss-Seidel with a discrete linear operator
void DDDOperatorJacobi(op,m,fin,fout) DDDOperator *op; DDDFunction *fin,*fout; int m;	Applies a Jacobi iteration with a discrete linear operator.
DDDomainSolver *DDDomainSolverCreateWithCommandLine(domain,argc,args) DDDomain *domain; int *argc; char **args;	Creates DDDomainSolver context for use in solving a PDE on a single domain. Uses either DDMultiGrid or DDOneGrid solver depending on command line option -multigrid or -onegrid
void DDDomainSolverUseInitialGuess(ds) DDDomainSolver *ds;	Forces the DDDomainSolver to use whatever is in the approximate solution as an initial guess if an iterative solver is used.
DDOneGrid *DDOneGridCreateWithCommandLine(domain,argc,args) DDDomain *domain; int *argc; char **args;	Creates DDOneGrid context for use in solving a PDE on a single domain
DDOneGrid *DDOneGridCreate(domain,svmethod) SVMETHOD svmethod; DDDomain *domain;	Creates DDOneGrid context for use in solving a PDE on a single domain.
void DDOneGridSetUp(og) DDOneGrid *og;	Called after call to DDOneGridCreate() but before call to DDOneGridSolve()

DDDFunction *DDOneGridSolve(og) DDOneGrid *og;	Solves PDE on domain.
void DDOneGridDestroy(og) DDOneGrid *og;	Frees space used by the onegrid solver.
void DDOneGridAddLineGraph(og,win) XBWindow win; DDOneGrid *og;	Adds window to plot line graph of residual and possible error.
void DDOneGridAddSolution(d,s) DDOneGrid *d; DDFunction *s;	If one has a function for the PDE solution this may be added to the DDOneGrid structure in order to calculate errors, etc. Use DDOneGridAddDSolution() for discrete solutions.
void DDOneGridSaveProblem(og,name) char *name; DDOneGrid *og;	Saves to a file the matrix, the RHS and the solution. This problem may be read in on a parallel machine. Call before or after a call to DDOneGridSolve().
DDMultiGrid *DDMultiGridCreateWithCommandLine(domain,domains,argc,args) DDDomain *domain,**domains; int *argc; char **args;	Creates DDMultiGrid context for use in solving a PDE on a single domain.
void DDMultiGridSetUpWithCommandLine(ctx,argc,args) int *argc; char **args; DDMultiGrid *ctx;	Called after a call to DDMultiGridCreateWithCommandLine(). Allocates memory for multigrid solves.
DDMultiGrid *DDMultiGridCreate(domain,indomains,l) DDDomain *domain,**indomains; int l;	Creates DDMultiGrid context for use in solving a PDE on a single domain
void DDMultiGridSetUp(ctx,itmethod) DDMultiGrid *ctx; ITMETHOD itmethod;	Called after call to DDMultiGridCreate() but before call to DDMultiGridSolve()
DDDFunction *DDMultiGridSolve(ctx) DDMultiGrid *ctx;	Solves PDE on domain.
void DDMultiGridDestroy(ctx) DDMultiGrid *ctx;	Frees space used by MultiGrid.
void DDMultiGridAddLineGraph(ctx,win) XBWindow win; DDMultiGrid *ctx;	Adds window to plot line graph of residual and possible error.
void DDMultiGridAddSolution(d,s) DDMultiGrid *d; DDFunction *s;	If one has a function for the PDE solution, this may be added to the DDMultiGrid structure in order to calculate errors, etc. Use DDMultiGridAddDSolution() for discrete solutions
void DDMultiGridUseJacobi(ctx,damp) DDMultiGrid *ctx; double damp;	Forces the multigrid solver to use Jacobi smoothing rather than Gauss-Seidel. This is intended mainly for comparison, there is usually no good reason to use Jacobi on a sequential machine.
void DDMultiGridUseGS(ctx) DDMultiGrid *ctx;	Forces the multigrid context to use Gauss-Seidel smoothing rather than symmetric Gauss-Seidel.
void DDDomainSolverAddWindow1(ctx, window, winfo) DDDomainSolver *ctx; XBWindow window; XBInfo *winfo;	Adds window to DDDomainSolver structure for plotting grid, solution, etc.
void DDDomainSolverUseSurfacePlot(ctx) DDDomainSolver *ctx;	Draws solutions, errors, etc. using a surface plot rather than the default contour plot
void DDDomainSolverAddWindow2(ctx, window, winfo) DDDomainSolver *ctx; XBWindow window; XBInfo *winfo;	Adds window to DDDomainSolver structure for plotting grid, solution, etc
void DDDomainSolverAddDSolution(ctx, solution) DDDomainSolver *ctx; DDDFunction *solution;	If one has a discrete function for the PDE solution, this may be added to the DDDomainSolver structure in order to calculate errors, etc. Use DDDomainSolverAddSolution for continuous solutions.
void DDDomainSolverSetHold(ctx, hold) DDDomainSolver *ctx; int hold;	Determines how long graphics calls by DDDomainSolver routines will hold
void DDDomainSolverAddSolution(ctx, s) DDDomainSolver *ctx; DDFunction *s;	Adds exact solution to domain solver context
void DDMultiGridAddWindow1(domain, window, winfo) DDMultiGrid *domain; XBWindow window; XBInfo *winfo;	Adds window to multigrid structure for plotting grid, solution, etc.
void DDMultiGridAddWindow2(domain, window, winfo) DDMultiGrid *domain; XBWindow window; XBInfo *winfo;	Adds window to multigrid structure for plotting grid, solution, etc

void DDMultiGridAddDSolution(multigrid, solution) DDMultiGrid *multigrid; DDDFunction *solution;	If one has a discrete function for the PDE solution, this may be added to the DDMultiGrid structure in order to calculate errors, etc.
void DDMultiGridSetNumberSmoothUp(multigrid,n) DDMultiGrid *multigrid; int n;	Sets the number of postsmoothing steps to use
void DDMultiGridSetNumberSmoothDown(multigrid,n) DDMultiGrid *multigrid; int n;	Sets the number of presmoothing steps to use.
void DDMultiGridSetCycles(multigrid,n) DDMultiGrid *multigrid; int n;	1 for V cycle, 2 for W cycle
double DDMultiGridUseAdditive(multigrid) DDMultiGrid *multigrid;	Uses additive form of multigrid rather than classical V or W cycle.
double DDMultiGridUseFull(multigrid) DDMultiGrid *multigrid;	Uses full multigrid as preconditioner.
void DDMultiGridSetHold(ctx, hold) DDMultiGrid *ctx; int hold;	Determines how long graphics calls by multigrid routines will hold
void DDMultiGridUseSurfacePlot(ctx) DDMultiGrid *ctx;	Draws solutions, errors using a surface plot rather than the default contour plot.
void DDOneGridAddWindow1(ctx, window, winfo) DDOneGrid *ctx; XBWindow window; XBInfo *winfo;	Adds window to DDOneGrid structure for plotting grid and approximate solution.
void DDOneGridUseSurfacePlot(ctx) DDOneGrid *ctx;	Draws solutions, errors, etc. using a surface plot rather than the default contour plot
void DDOneGridAddWindow2(ctx, window, winfo) DDOneGrid *ctx; XBWindow window; XBInfo *winfo;	Adds window to DDOneGrid structure for plotting error of approximate solution, etc.
void DDOneGridAddDSolution(ctx, solution) DDOneGrid *ctx; DDDFunction *solution;	If one has a discrete function for the PDE solution, this may be added to the DDOneGrid structure in order to calculate errors. Use DDOneGridAddSolution for continuous solutions
void DDOneGridSetHold(ctx, hold) DDOneGrid *ctx; int hold;	Determines how long graphics calls by DDOneGrid routines will hold.
void DDOneGridAddCoarseOneGrid(ctx, onegrid) DDOneGrid *ctx; DDOneGrid *onegrid;	If one is using the overlapping Schwarz method, use this routine to set the coarse domain solver to use.
void DDOneGridSetNumberSubdomains(ctx, n) DDOneGrid *ctx; int n;	Sets the number of subdomains to use for overlapping Schwarz.
void DDOneGridSetOverlap(ctx, n) DDOneGrid *ctx; int n;	Sets the overlap to use for overlapping Schwarz.
void DDOneGridSetUseAdditive(ctx) DDOneGrid *ctx;	Use additive Schwarz if the overlapping Schwarz method is used
void DDOneGridSetSubdomainMethod(ctx, method) DDOneGrid *ctx; SVMETHOD method;	Sets the method to use on each subdomain if the overlapping Schwarz method is used
DDSchwarz *DDSchwarzCreateWithCommandLine(domain1, domain2, argc, argv) int *argc; DDDomain *domain1, *domain2; char **argv;	Creates DDSchwarz context for use in solving a PDE on two domains
void DDSchwarzSetUpWithCommandLine(schwarz,argc,args) DDSchwarz *schwarz; int argc; char **args;	Allocates space, etc. for an alternating Schwarz solver.
void DDSchwarzSolve(schwarz) DDSchwarz *schwarz;	Solves the two domain problem using alternating Schwarz method.
void DDSchwarzAddWindow1(s,w) DDSchwarz *s; XBWindow w;	Adds window for displaying solution.
void DDSchwarzAddWindow2(s,w) DDSchwarz *s; XBWindow w;	Adds window for displaying error
DDDomainSolver *DDSchwarzGetDomainSolver1(s) DDSchwarz *s;	Returns the domain solver for the first subdomain.
DDDomainSolver *DDSchwarzGetDomainSolver2(s) DDSchwarz *s;	Returns the domain solver for the second subdomain

void DDGridUseChacoPartitioner(grid) DDGrid *grid;	Sets the grid to use the Chaco package for grid partitioning. Works only for the unstructured grids
int DDGridChacoHelp(argc,args) int *argc; char **args;	Prints help message for Chaco package to stderr
void DDGridUseChacoPartitionerWithCommand-Line(grid,argc,args) DDGrid *grid; int *argc; char **args;	Parses command line to set various arguments for Chaco partitioner. By default this uses a multilevel spectral partitioner.
void DDGridUsePamPartitioner(grid) DDGrid *grid;	Sets the grid to use the Pam package for grid partitioning. Works only for the unstructured grids.
int DDGridPamHelp(argc,args) int *argc; char **args;	Prints help message for Pam package to stderr.
void DDGridUsePamPartitionerWithCommandLine(grid, argc,args) DDGrid *grid; int *argc; char **args;	Parses command line to set various arguments for Pam Partitioner. By default this uses a multilevel spectral partitioner
void DDGridUseNaivePartitioner(grid) DDGrid *grid;	Force grid to be partitioned using simple naive scheme.
void DDGridUseNaivePartitionerWithCommandLine(grid,argc,args) DDGrid *grid; int *argc; char **args;	Force grid to be partitioned using simple naive scheme.
void DDGridUseBaSiPartitioner(grid) DDGrid *grid;	Sets the grid to use the BaSi package for grid partitioning. Works only for the unstructured grids
int DDGridBaSiHelp(argc,args) int *argc; char **args;	Prints help message for BaSi package to stderr
void DDGridUseBaSiPartitionerWithCommandLine(grid, argc,args) DDGrid *grid; int *argc; char **args;	Parses command line to set various arguments for BaSi partitioner. By default this uses a multilevel spectral partitioner.
void DDGridStore(grid,name) DDGrid *grid; char *name;	Stores a grid to file.
DDGrid *DDGridLoad(name) char *name;	Loads a grid from file
void MGMCycle(mglevels) MG **mglevels;	Given an MG structure created with MGCreate(), runs one multiplicative cycle down through the levels and back up.
MG **MGCreate(levels) int levels;	Creates a MG structure for use with the multigrid code.
void MGDestroy(mg) MG **mg;	Frees space used by an MG structure created with MGCreate().
int MGCheck(mg) MG **mg;	Checks that all components of MG structure have been set. Use before MGCycle()
void MGSetNumberSmoothDown(mg,n) int n; MG **mg;	Sets the number of presmoothing steps to use on all levels. Use MGSetSmootherDown() to set it differently on different levels.
void MGSetNumberSmoothUp(mg,n) int n; MG **mg;	Sets the number of post smoothing steps to use on all levels. Use MGSetSmootherUp() to set it differently on different levels
void MGSetCycles(mg,n) int n; MG **mg;	Sets the number of cycles to use. 1 is V cycle; 2 is W cycle. Use MGSetCyclesOnLevel() for more complicated cycling.
void MGACycle(mg) MG **mg;	Given an MG structure created with MGCreate() runs one cycle down through the levels and back up. Applies the smoothers in an additive manner.
void MGFMG(mg) MG **mg;	Given an MG structure created with MGCreate(), runs full multigrid.
void MGCycle(mg,am) MG **mg; int am;	Runs either an additive or multiplicative cycle of multigrid.
void MGSetCoarseSolve(mg,f,c) MG **mg; void (*f)(); void *c;	Sets the solver function to be used on the coarsest level

void MGSetResidual(mg,l,f,c) MG **mg; void (*f)(); void *c; int l;	Sets the function to be used to calculate the residual on the lth level.
void MGSetInterpolate(mg,l,f,c) MG **mg; void (*f)(); void *c; int l;	Sets the function to be used to calculate the interpolation on the lth level.
void MGSetZeroVector(mg,l,f,c) MG **mg; void (*f)(); void *c; int l;	Sets the function to be used to zero a vector on the lth level.
void MGSetRestriction(mg,l,f,c) MG **mg; void (*f)(); void *c; int l;	Sets the function to be used to restrict vector from lth level to the l-1 level.
void MGSetSmootherUp(mg,l,f,c,d) MG **mg; void (*f)(); void *c; int l,d;	Sets the function to be used as smoother after coarse grid correction (postsmoother).
void MGSetSmootherDown(mg,l,f,c,d) MG **mg; void (*f)(); void *c; int l,d;	Sets the function to be used as smoother before coarse grid correction (presmoother).
void MGSetCyclesOnLevel(mg,l,n) MG **mg; int l,n;	Sets the number of cycles to run on this level
void MGSetRhs(mg,l,c) MG **mg; void *c; int l;	Sets the vector space to be used to store right-hand side on a particular level. User should free this space at conclusion of multigrid use.
void MGSetX(mg,l,c) MG **mg; void *c; int l;	Sets the vector space to be used to store solution on a particular level.User should free this space at conclusion of multigrid use.
void MGSetR(mg,l,c) MG **mg; void *c; int l;	Sets the vector space to be used to store residual on a particular level. The user should free this space at conclusion of multigrid use

Bibliography

[1] William D. Gropp and Barry F. Smith. Simplified Linear Equation Solvers users manual. Technical Report ANL-93/8, Argonne National Laboratory, March 1993.

[2] William D. Gropp and Barry F. Smith. Users manual for KSP: Data-structure-neutral codes implementing Krylov space methods. Technical Report ANL-93/30, Argonne National Laboratory, August 1993.

[3] William D. Gropp and Barry F. Smith. Scalable, extensible, and portable numerical libraries. In *Proceedings of Scalable Parallel Libraries Conference*, pages 87–93. IEEE, 1994.

Function Index

D

X