December 31, 1997

DIME

Distributed Irregular Mesh Environment

Roy Williams

California Institute of Technology

For having first to civilise a space
Wherein to play your violin with grace.

— GWENDOLYN BROOKS

Contents

1.0 What DIME is

The Distributed Irregular Mesh Environment[1] is a programming environment for creating, manipulating and performing calculations on an unstructured triangular mesh. The mesh covers a two-dimensional manifold, whose boundaries may be defined by straight lines, arcs of circles, or Bezier cubic sections.

The same code runs on a Unix workstation or a MIMD distributed-memory parallel processor.

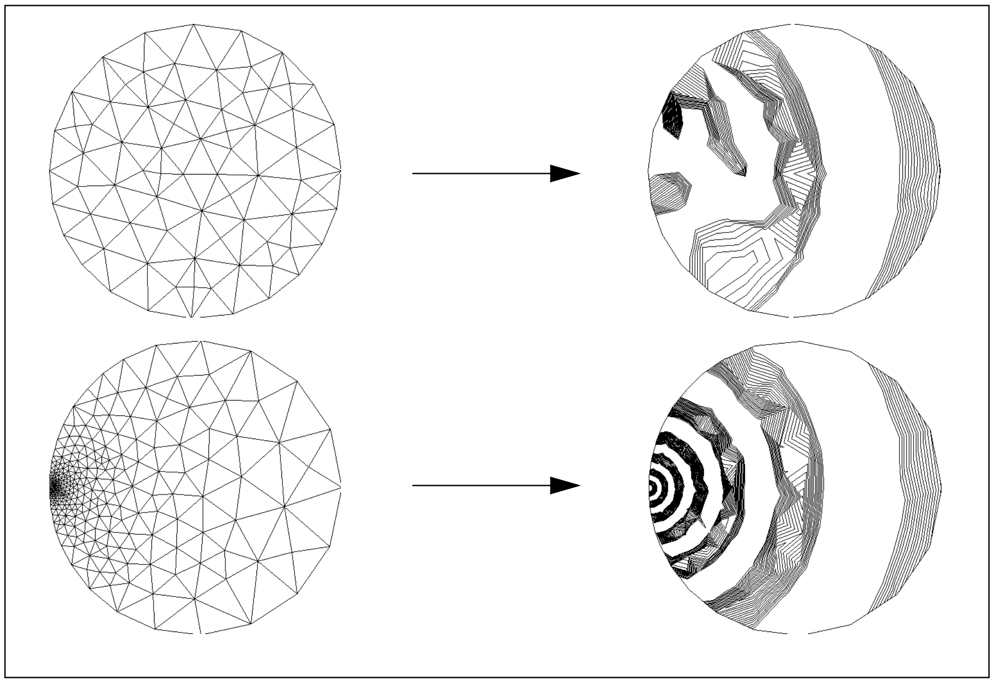
A coarse mesh is interactively created with a sequential machine using the Delaunay triangulation[2]; this mesh is sufficient to resolve the topology of the model boundary. The coarse mesh is loaded into a single processor of the distributed machine, then dynamically load-balanced and adaptively refined as the calculation progresses.

DIME has been used for transonic flow simulations[3,4], variational calculations with quadratic finite elements[5], Monte-Carlo simulations of liquid-membrane theory[6], and Boundary Element calculations of the electrical properties of certain fish[7].

1.1 Triangular Meshes and Refinement

Figure 1 shows two unstructured triangular meshes covering a disk, and contour plots of a

Figure 1 Two triangular meshes and resulting contour plots



function at the resolution of each mesh. The lower right plot shows more detail of the function because the mesh is adapted at the place where the function is complex, revealing the detail there. The idea of DIME is to use the mesh to discretize space, with the discretization at higher resolution where necessary. The higher resolution may be put in at the start or adaptively as the calculation progresses.

The mesh consists of a set of triangles or *elements*, and *nodes*, which are the points where several elements meet. Elements and nodes are the basic data structures of DIME, and a user application for DIME generally consists of loops over all the elements which take data from their neighboring nodes, store it and put it back in the nodes, or vice versa with loops over the nodes.

1.2 Programming Environment

The control of the mesh by a user is achieved by writing a C program, consisting of two parts: structure definitions and a set of functions. The functions are associated with menu buttons, so that the program may be linked with the DIME library and run interactively. The program may also be run in batch mode with a script file.

Programming constructions are macros such as `FORALLNODES(node) ... NEXTNODE(node)`, which is a loop over all the nodes of the mesh.

The user may access and modify data by code such as `node->user->pressure = p;` which is an assignment to the variable `pressure` in the data space of the node `node`.

Refinement is accomplished with code such as

```
FORALLELMTS(elmt)
   set_refine (elmt, <truth value>);
NEXTELMT(elmt)
refine();
```

which is a loop over all elements, and the truth value is `TRUE` or `FALSE` depending on whether that element is to be refined or not. The parallel call to `refine()` actually refines the mesh, interpolating the user data to put sensible values into the user data space of new nodes or elements which may be created.

Load balancing for a parallel computer may be achieved in the same way with calls to `set_balance` to give each element a new processor in which to reside, then a parallel call to `balance()` to move the mesh between processors.

Since it is rather difficult to assign new processors to the elements in parallel, a function `balance_orb` is provided to balance the mesh by Orthogonal Recursive Bisection. See the manual pages for more details.

1.3 What DIME is Not

At present DIME is not able to remove nodes and elements from a mesh: the application should start with a relatively coarse mesh and compute with it, refine it and load-balance it until the mesh is sufficiently detailed. DIME cannot do for example a moving front problem, where the mesh should be fine at the front and coarse elsewhere, since as soon as the front moves, the mesh would have to coarsen once the front has passed.

DIME is not good at moving mesh problems such as Lagrangian fluid dynamics[8], where the nodes of the mesh move in some way. If a node moves outside the group of elements which surround it, the mesh will be ‘tangled’ and the code may crash. Movement which does not violate this constraint is possible; however the topological relaxation process (Section 9.2) must be regularly applied if the mesh is to keep moving. This is fine sequentially, but in parallel this involves moving elements from processor to processor, and is thus rather inefficient.

DIME is restricted to two dimensional meshes, though these may be embedded in some higher dimensional space. A future version of DIME will have the ability to distinguish between different regions of a manifold, and to deal with non-planar two-dimensional topologies such as a cylinder or torus.

I hope to extend the concepts and methods of this manual to a three-dimensional version of DIME for tetrahedral meshing of complex geometries. Most of the difficulty here is the definition of a 3D boundary, which is considerably more complex than the definition of a 2D boundary, and also that the graphics tools must be much more sophisticated for visualization of 3D phenomena.

2.0 Getting Started

2.1 Hardware

DIME runs in parallel on the computer systems which support the Express parallel programming environment[9], and it should run sequentially on any machine which runs UNIX. This has been verified for Sun workstations, PC's with Xenix, the Macintosh II, and the Cray Unicos environment.

The parallel systems supported by Express are (at date of writing):

- Transputer boards from Definicon, Levco, Meiko, MicroWay, Quintek
- Meiko Computing Surface
- NCUBE and NCUBE/2 systems
- Symult S2010
- Up to 8 Cray processors

You will also need a mouse and a color graphics display. For hard-copy, a PostScript[10] printer is needed.

2.2 Software

To run a DIME program in parallel, the Express parallel programming environment should be installed.

To run sequentially, you need a device which supports the ParaSoft Plotix[9] graphics package, which is rather elementary but quite portable. You may of course run on a single processor of a parallel machine with Express. Plotix can run on

- An X-windows server
- A Sun workstation under Sunview
- A Tektronix 4105 or emulator
- The Macintosh II
- A PC with an Enhanced Graphics Adapter

You may also use DIME without Express by linking with either the Sunview or X-windows Plotix libraries supplied with DIME, and running on a Sun workstation or X server.

2.3 Installation

Make an empty directory called `dime` and copy in all the DIME source code. You should add the following two lines to your `.cshrc` file:

```
setenv DIME '<directory name>'
set path = ($DIME/bin $path)
```

where <*directory name*> is the name of the directory into which you copied the source code.You may need to change some of the file names in the file `Makefile.defs` which define where the various compilers and the Plotix library live. The default configuration is for an X-windows version of DIME. You may now make the source code for your workstation by typing

```
make seq
```

which make the sequential tools `curvetool` and `meshtool`, the sequential DIME library, and the examples. For parallel machines you may type

`make C`	for Cray systems
`make N`	for NCUBE systems
`make S`	for Symult systems
`make T`	for Transputer systems

This will make the DIME library and the examples for your parallel machine.

An executable which runs on the workstation has the suffix `.seq`, and one for your parallel machine has the suffix `.C`, `.N`, `.S`, or `.T` as the case may be.

3.0 Running a DIME program

Assuming that the DIME source code and examples are installed (see Section 2), go to the examples directory `$(DIME)/examples/simple`, and type `simple.seq`. You should see a graphics window appear, as shown in Figure 2, with a blank drawing area to the right, and a menu area to the left. The window in which you invoked the program is a command area, in which you may type text and in which messages appear from the code. DIME announces that it is running, and the number of processors which are in use, in this case one.

You are now ready to run a program which can read a boundary definition and mesh from a file, refine it, and draw a contour plot of a function on that mesh. The program can also adaptively refine the mesh according to the complexity of the function it is contouring, and load-balance the work among any number of processors.

3.1 Reading in a Mesh

First we need to read in the definition of a domain which is to be meshed, and a coarse mesh which has been created over it. The program `meshtool` is used to interactively create a coarse mesh on a domain, and is described in Section 7.

Click the mouse button on the menu item “DIME”, and the menu display shows a new list of menu items. Click “Readmesh”, and the prompt `Boundary File name?` appears in the command window. Reply with the name `simple`; the system adds the default suffix `.bdy` to this name, looks for the file `simple.bdy`, and reads it in. You are prompted for a mesh file in the same way: reply `simple` again, and the mesh file `simple.mesh` is read. You should see a triangular mesh appear in the drawing area. The bounding box of the mesh is printed in the command region: in this case (0,0) to (1,1); it is a unit square.

3.2 Menus

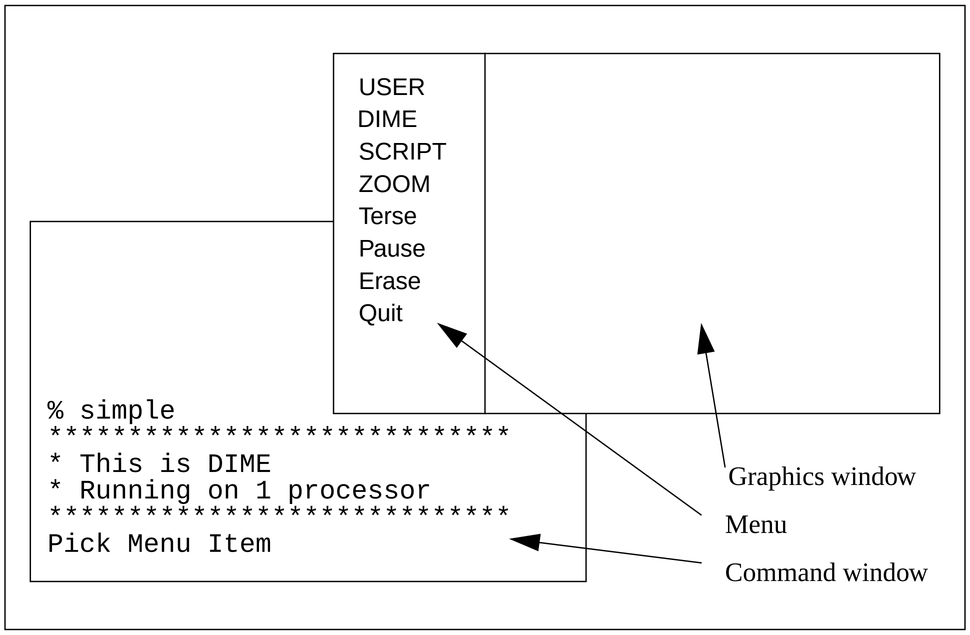
DIME menus are arranged in a tree structure. Each menu item corresponds to either a submenu or a function in the program. These functions and submenus may have been defined either by DIME or by the user. Thus we may think of the code that is running as a dictionary of functions, each of which may be accessed by a menu button. The default DIME part of the menu tree with its functions is described after the References. The user part of the menu tree is of course determined by the user, and may be selected by clicking “USER” in the root menu.

By convention, submenu names are all capitals, and items that cause other action lower-case with initial capitals.

The root menu is that shown in Figure 2; all the submenus look similar except for an extra entry at the top “TO_ROOT” which escapes from the submenu back to the root menu.

You might like to try moving around the menu tree, and seeing what some of the items do.

Figure 2



```
% simple
*****************************
* This is DIME
* Running on 1 processor
*****************************
Pick Menu Item
```

3.3 The Logical Mesh

There is a more abstract view of the mesh which shows the internal structure better. First erase the first view of the mesh from the screen, by clicking “TO_ROOT” to get back to the root menu, then “Erase” to erase the screen, then “DIME” to get back to the DIME menu. Now click “Logicalmesh”.

Instead of just showing the edges of the mesh, we see red triangles, which are the elements, connected to nodes, which are the star-shaped collections of lines between the elements.

3.4 Refinement

Click the menu item “Rectrefine”, and you are prompted for the desired mesh spacing (try 0.2), and to define a rectangle by clicking the mouse at first at one corner, then the diagonally opposite corner. Each element whose center is within the chosen rectangle, and which has a side longer than the desired mesh spacing, is refined. The process continues until each edge of each of the elements in the rectangle is shorter than the chosen mesh spacing. We can thus refine the mesh explicitly by choosing a mesh spacing and a region.

3.5 The User Program

Go back to the root menu by clicking “TO_ROOT”, then to the “USER” menu. This is the part of the code specific to the user code, and the functions you find here are defined in the program `simple.c` in this directory, rather than part of the general DIME function library. In Section 10 we shall see how to write such programs.

Click “Contour”, and a contour plot should appear of a function which is smooth at the top right of the screen, and more complex to the bottom left. If you refined the mesh earlier to a spacing of 0.2, the bottom left part of the function will be quite indistinct because the mesh spacing is not fine enough. Try clicking “Adaptrefine” and “Contour” alternately. The mesh is only refined at the lower left corner where it needs to be, and not at the top right, where there is already sufficient resolution to see the function.

As you keep on refining the mesh, everything starts to slow down as the machine has more and more mesh to keep track of and contour. Perhaps we should try using a parallel machine, if one is available.

3.6 Parallel Operation

There should also be a parallel version of the `simple` code in the examples directory. We assume here that Express is properly installed on your system, that the machine which you are logged on to is a host to a parallel machine that runs Express, and that the parallel machine is properly booted and available, and able to draw on your screen. If you are using a transputer machine, the parallel executables have the suffix `.T`; for other machines there are different suffices. The Cubix server is part of Express, and is used like this to download and start a distributed application such as `simple.T`:

```
% cubix -d2 -TX simple.T
```

where the `-TX o`ption indicates that the graphics is to be sent to an X-windows server`;` the option `-Tsun` is for Sunview graphics, and `-Tega` for a PC with EGA screen. `The option -d2` option indicates the logarithm to the base two of the number of processors to be used; in this case we use four processors.

DIME only runs if the number of processors is a power of two: this is because of the Orthogonal Recursive Bisection method for load balancing, which splits the mesh into two, then each half into two, then each quarter into two, etc.

The only difference in this case from the sequential use of DIME is that the banner announces that DIME is running on four processors rather than one processor. There may also be some messages before the DIME banner, which are from Express: typically a sequence of letter ‘b’s, one for each 512 bytes as the executable code is downloaded to the processors of the parallel machine, and a message ‘Loaded, starting’.

We may continue as before. Click “DIME” then “Readmesh”, then reply `simple` to both prompts. Again the mesh is read in, but this time to only one processor of the machine. We could carry on as before refining and contouring, but we would only be using one processor, while the other three are idle.

3.7 Load Balancing

The extra ingredient for parallel operation is load balancing, which is the splitting of the mesh into more-or-less contiguous groups of triangles, with approximately equal numbers in each group, one group for each processor.

Go to the DIME menu and click “Balance”. In the command window should appear:

```
Splitting on channel 0 *****
Splitting on channel 1 **
```

Now erase the screen (“TO_ROOT Erase” as before), and click “Drawmesh” in the DIME menu. The mesh is redrawn, but with two differences. First the edges of the mesh are not all blue, but some are red, where the red edges divide the mesh into four quadrants. A red edge is a division between processors: each quadrant corresponds to the ‘domain’ of one of the four processors executing the code. There is also a more subtle difference from the sequential version; the drawing of the mesh is done processor by processor. First the domain of one processor is drawn, then that of the next, and so on.

A view of the logical structure of the mesh can be obtained with the “Logicalmesh” menu option. You might want to erase the screen first, using the sequence “TO_ROOT Erase”. This view of the mesh shows the divisions between the processor domains more clearly.

Now to do some real computation. Go to the USER menu (via clicking “TO_ROOT” in the DIME menu), and try “Adaptrefine” and “Contour” a few times. Notice that one processor (at the lower left) is getting all the new mesh, because that is where most of the refinement is occurring. We need to redistribute the mesh more fairly among the processors, using the “Balance” function (in the DIME menu).

Try repeating the sequence “Adaptrefine TO_ROOT DIME Balance TO_ROOT USER Contour” a few times. You might want to include the function “Memory” from the DIME menu which gives an inventory of the memory consumption of each processor. Try running through this cycle a few times. As the memory usage increases, a virtual memory machine such as a workstation will find extra memory, but the disk will grind a lot as the swapping occurs. With a parallel machine though, one processor will eventually run out of memory; it can now do nothing but call a halt to the entire program by broadcasting an abort signal.

You may notice that the graphics is considerably slower with the parallel machine than when the program runs on a workstation, because the graphics commands must make their way through the parallel machine, out to the host machine, perhaps through a network and eventually to the screen.

3.8 Scripts

While it is good to have an interactive menu system to run the code and see the results on the screen, one would also like the ability to encapsulate interactive sessions into a file for later playback. The script mechanism provides the following capabilities:

- To run the application without the graphics display, either because it is not wanted or because it is not available.
- To run in batch mode without human interaction.

- To encapsulate frequently-used sequences of menu choices.

A script may be invoked or created from the menu, it may be altered or extended with a text-editor, or it may be used on the command line when executing the application. The steps outlined above are contained in a script called `simple.script`. You can use this either by typing

```
% simple.seq -f simple
```

for the sequential version, or

```
% cubix -d2 -TX simple.T -f simple
```

for the parallel version. You may also execute the code as before with no command-line arguments, then click “SCRIPT” and “Use_script” as soon as the graphics window is available.

You may make a script file by clicking “Make_script” in the SCRIPT menu, and from that point on all menu commands and user input is logged to the script file until you click “End_script”.

A script file is just a listing of the menu labels that were clicked with any typed input included, and may be adjusted or created with a text editor.

DIME understands the `-f` switch on the command line to run from a script, and also the -g switch:

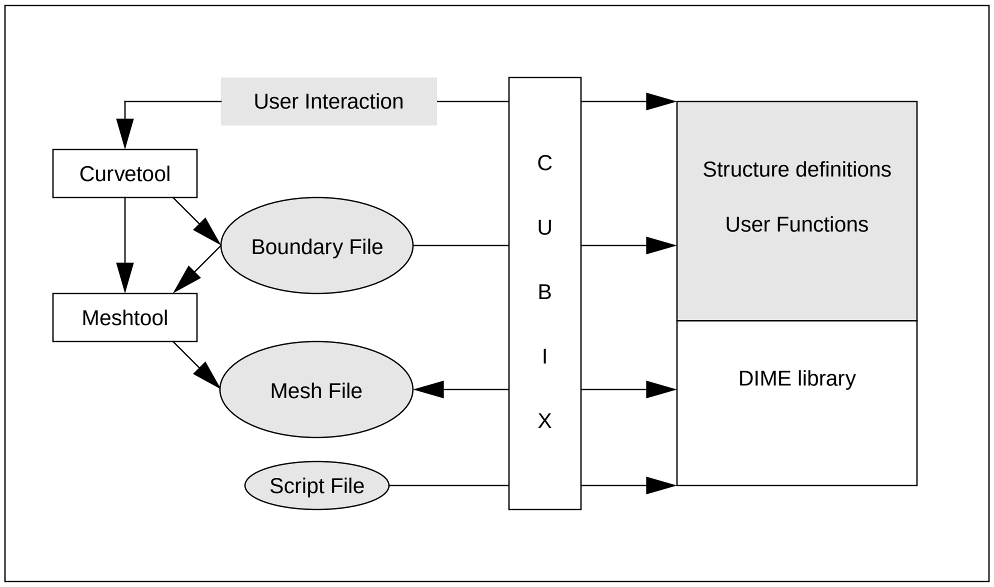
- `-g2` ⇒ run with graphics and menus
- `-g1` ⇒ run with graphics and no menus (only with a script file)
- `-g0` ⇒ run with no graphics (only with a script file)

Anything else on the command line is passed to the user code (see Section 10.0).

4.0 The Components of DIME

Figure 3 shows the structure of a DIME application. The shaded parts represent the contribution from the user, being a definition of a domain which is to be meshed, a definition of the data to be maintained at each element, node and boundary edge of the mesh, and a set of functions which manipulate this data. The user may also supply or create a script file as explained in Section 3.8.

Figure 3 Major Components of DIME



The first input is the definition of a domain to be meshed. A file may be made using the format described in Section 5, or may be made by the sequential program `curvetool`, which allows straight lines and cubic splines to be manipulated interactively to define a domain. This program is described in Section 6.

Before sending a domain to a DIME application, it must be predigested to some extent, with the help of a human. The user must produce a coarse mesh which defines the topology of the domain to the machine. This is done with the sequential program `meshtool`, which allows the user to create nodes and connect them up to form a triangulation.

The user writes a program for the mesh, and this program is loaded into each processor of the parallel machine. When the DIME function `readmesh()` is called, or “Readmesh” clicked on the menu, the mesh created by `meshtool` is read into a single processor of the machine, and then the function `balance_orb()` may be called (or “Balance” clicked on the menu) to split the mesh into domains, one domain for each processor.

The user may also call the function `writemesh()` or click “Writemesh” in the menu, which causes the parallel mesh to be written to disk. If that mesh is subsequently read in, it is read in its domain-decomposed form, with different pieces assigned to different processors.

4.1 Cubix and DIME

The Cubix server[9,11] is part of Express[9] and it is this which makes it easy to write a parallel application such as DIME ***which also runs on a sequential machine***. The parallel machine works by having each processor send messages either to other processors or to the host machine; it is this host which carries on a dialogue with a user. Parallel programs are often written in two parts, being a processor program and a host program. The host program is responsible for access to peripherals such as the graphics screen, disks, printers, etc., and may be tailor-made for the particular application running on the array of processors.

Cubix takes a different approach, which is to make the host program an invisible general-purpose server, capable of opening and closing files, starting up other processes on the host machine, or drawing on the user’s graphics screen. When a processor wishes to print something on the user’s screen, it simply calls `printf()`, the usual C printing function, rather than sending a message to the host, which would be interpreted and would cause the print to occur.

A Cubix parallel code thus looks very much like a sequential program, except that the program is running in each processor of the parallel machine; we may think of a sequential program as one running on a 1-processor parallel machine.

Since it is not possible to program the host with a Cubix program, the programmer is forced to make everything parallel, so that sequential bottlenecks are eliminated.

This section is not meant to give a complete description of Cubix; for more details see the Express literature[9].

4.2 I/O Modes

There are two ways for I/O such as `printf()` and `scanf()` to occur in DIME, being *single* and *multiple* mode.

Single mode is when all the processors are to send and receive the same data, for example in the following dialogue with the user:

```
% How Many Iterations to run?
% 20
% What is the Tolerance?
% 0.001
```

Each processor calls `printf()` to print the prompt message, but it only appears once, and each processor receives the same values (20 and 0.001) when it calls `scanf()` to get the response. This is the default mode for DIME, and the DIME functions expect all open file pointers to be in this mode on entry to a function.

Multiple mode is for I/O where each processor sends or receives its own value. For example four processors may produce a memory summary:

```
Processor 0: Available memory 1278932 bytes
Processor 1: Available memory 677892 bytes
Processor 2: Available memory 1046722 bytes
Processor 3: Available memory 1133457 bytes
```

and the code for this would be

```
fmulti(stdout);
printf(“Processor %d: Available memory %d bytes\n”,
   procnum, memory);
fsingl(stdout);
```

The variable `procnum` is an external variable, and is an identifying number telling the program which processor it is running on. The call to `fmulti` changes the mode of the standard output stream to multiple mode. Notice the return to the default single mode after the print statement. For more details on I/O modes see the Cubix literature[9].

4.3 Loosely Synchronous Programming

DIME programs run *loosely synchronously*[11]. This means that when certain functions are called (those which involve communication), all the processors of the machine must call the same function. This is also called Barrier synchronization. An example is the `balance()` function, which splits the mesh into equal pieces for the processors. This is obviously a cooperative venture, and if any processors are not involved, the others will be waiting for messages that never arrive and the machine deadlocks.

This does not mean that the processors must call the function at the same wall-clock time, just that if one processor calls that function, all the others must eventually do so.

When writing to a file pointer in multiple mode, each processor is saving up the bytes produced in a buffer; only when the loosely synchronous `fflush` call is made does the data make its way to the user. Similarly with graphics: each processor stores up the graphics until the loosely synchronous `usendplot` call is made, which flushes the graphics output to the screen. See the Express documentation for more details.

4.4 Reading and Writing Meshes

A DIME program reads a mesh file which was produced either by the meshing program `meshtool` or by a DIME program. A mesh file carries with it the number of processors which produced it, and can only be read by a machine with that number or more processors. Since `meshtool` is a sequential program, its meshes may be read by a parallel machine with any number of processors. If a DIME program is running on P processors and writes out its mesh, that mesh file may only be read by a machine with P or more processors.

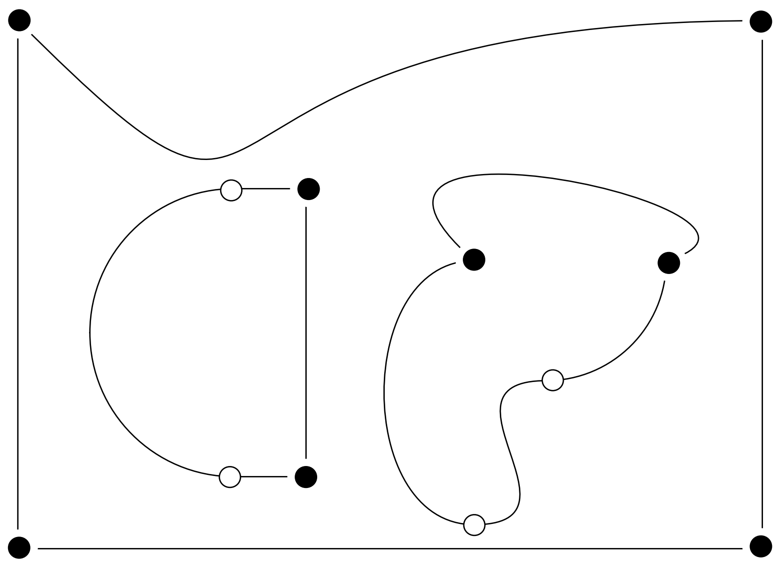
When a parallel machine reads a mesh which was written by P processors, it is read into the first P processors of the machine. For meshes made by `meshtool`, P is 1, and the mesh must not be too large to fit into a single processor of the machine. `meshtool` is intended to make a mesh which has enough nodes and elements to correctly represent the topology of the domain being meshed, with the serious refinement to a computational mesh done in parallel once this coarse mesh has been divided up among the processors.

When a mesh is read or written, the user has the opportunity to read or write her own user data to be associated with the nodes, elements and boundaries of the mesh, using the callback functions `user_read` and `user_write`, as described in the manual pages and Section 10.6.

5.0 Boundaries and Domains

Figure 4 shows an example of a DIME boundary structure. The filled blobs are *Points*, with *Curves* connecting the points. Each Curve may consist of a set of curve segments, shown in the Figure separated by open circles. The curve segments may be straight lines, arcs of circles, or Bezier cubic sections.

Figure 4 Boundary Structure



5.1 Boundary file format

Each of the Points and Curves may have a name, which may be used by the DIME code to assign boundary conditions or other information during the simulation.

There is a file with suffix `.bdy` which defines the Points and Curves of a boundary. This file may be produced with the interactive program `curvetool` or with a text editor. This file is read in by `meshtool` which creates a coarse triangulation for the specified domain, and outputs a file with the suffix `.mesh`.

A boundary file consists of a set of tokens separated by white space characters (space, tab or newline). A set of tokens begins with a keyword, which may be:

- ‘‘Point’’: following this is a Point name and a pair of numbers, being the coordinates of the Point.
- ‘‘Curve’’: following this is a Curve name, the name of the Point at the beginning of the Curve, a set of curve segments, and the name of the Point at the end of the Curve.

- ‘‘End’’: signifies the end of the boundary file. This should be the last token in the file.

For specifying curve segments there is a stack of numbers and the notion of a current point. At the beginning of a Curve specification, the coordinates of the first Point of the Curve are pushed on to the stack. Whenever a number is encountered during Curve specification, it is put on the stack. Interspersed with the numbers may be tokens which create curve segments. When the segment has been created, the stack is cleared and the coordinates of the end of the newly-created segment are put on the stack. The segment creation tokens are:

- “Lineto’’: The stack should contain 4 numbers, which are interpreted as the coordinates of the beginning and end of a straight line.
- ‘‘Arc’’: The stack should contain 5 numbers, interpreted as the coordinates of the center of an arc of a circle, the radius of the arc, and the start and end angles of the arc, measured anticlockwise from the x-axis. The arc is described anticlockwise.
- ‘‘Arcn’’: Same as Arc except the arc is described clockwise.
- ‘‘Curveto’’: Eight numbers are taken from the stack, being the coordinates of four control points x0, y0, x1, y1, x2, y2, x3, y3 for a Bezier cubic section[10]. The resulting curve is tangent to the line (x0, y0) - (x1, y1) at (x0, y0), and tangent to (x2, y2) - (x3, y3) at (x3, y3), and is guaranteed to stay inside the convex hull of the four points.

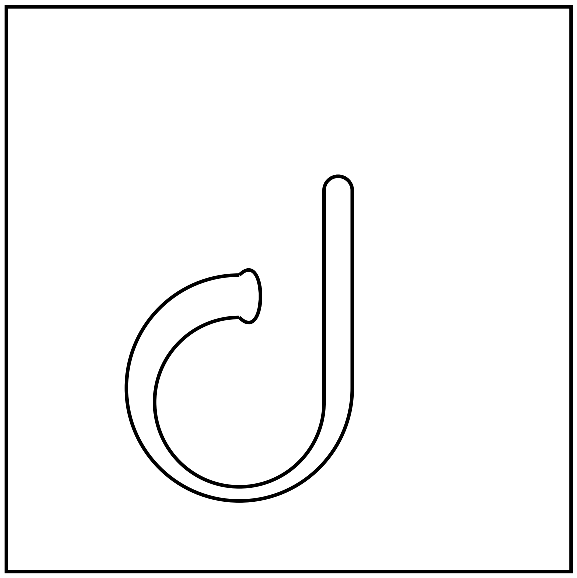
Following is an example of a boundary description file, which is a square with a hook shaped hole in it:

```
Point lower_left -40 -40
Point upper_left -40 40
Point upper_right 40 40
Point lower_right 40 -40
Point lower_corner -7 -4
Point upper_corner -7 2

Curve left lower_left -40 40 Lineto upper_left
Curve upper upper_left 40 40 Lineto upper_right
Curve right upper_right 40 -40 Lineto lower_right
Curve lower lower_right -40 -40 Lineto lower_left

Curve t1
   lower_corner
   -3 -8 -3 6 -7 2 Curveto
   upper_corner

Curve t2
   upper_corner
   -7 -14 16 90 0 Arc
```

```
    9 14 Lineto
    7 14 2 0 180 Arc
    5 -16 Lineto
    -7 -16 12 0 90 Arcn
    lower_corner
End
```

The boundary defined by this file is shown below.

Figure 5 Boundary ‘‘Hook’’ Defined Above



There are some restrictions on boundary definitions:

- A Curve may not begin and end at the same Point.
- Curves may not have zero length.

6.0 Curvetool

The program `curvetool` is for the interactive production of boundary files. With `curvetool` you may create Points and connect them together with Bezier cubic sections and straight lines..

Figure 6 Points and Curves in Curvetool

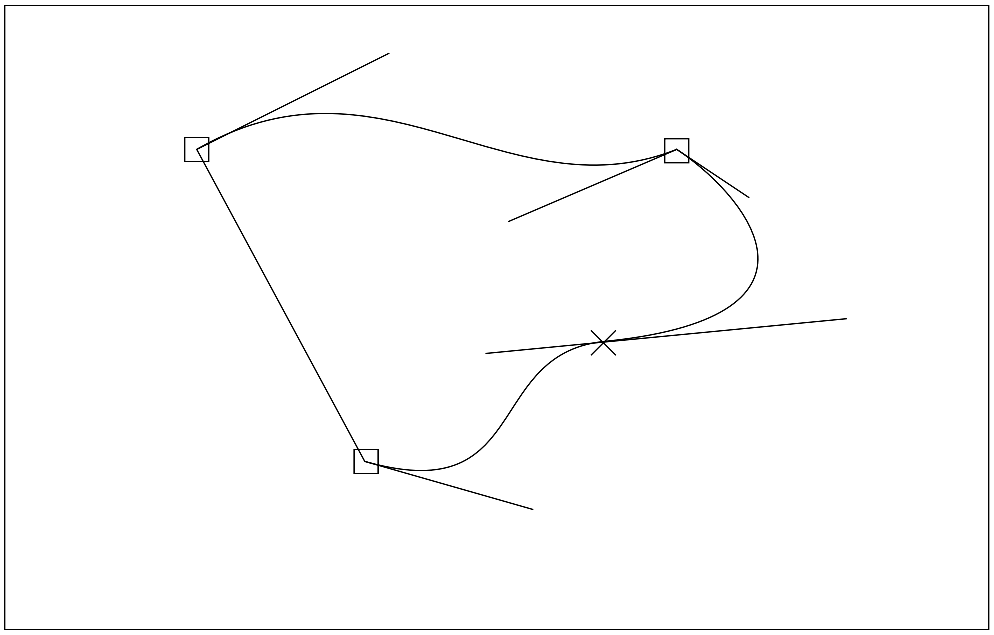


Figure 6 shows three Points, shown as squares, with three lines connecting them. One is a straight line, another is a Bezier section, and the third is a pair of Bezier sections with continuity and continuity of slope between them. At the junction of the two sections is a cross, which is a *section continuity mark* or SCM.

Each end of a cubic section is drawn with a tangential line at the end. The ends of these lines are the two *knots* for the section, so that a section is defined by four points: two end Points and two knots. The section is tangent to the line to the knot and contained in the convex hull of its four defining points. The lines from an SCM to its two knots must then be 180 degrees apart to ensure continuity of slope at the SCM.

A Point may be created with the menu option “Addpoint”, then click at the place where the Point should be. Two Points may be connected by clicking “Connect”, then selecting the two Points to be connected by clicking near them with the mouse.

A connection between Points is initially made as a straight line. This is not quite correct: `curvetool` treats a straight line as a degenerate cubic section: it is only when the boundary file is written that `curvetool` checks to see if the section is actually straight, in which case it is written to the file as a line rather than a cubic section. The straight-line connection between two Points actually has the two control points hidden under the line at one third and two thirds of the distance along the line.

A curve may be adjusted by clicking the option "Adjust", which fills the menu area with color and puts curvetool into Adjust mode. Clicking inside the colored menu area puts `curvetool` back into the normal menu mode. While in Adjust mode, you may select a Point, knot or section continuity mark, and move it. The first click finds the closest of these entities to the place you clicked and highlights the entity in red. The second click is the place to which the entity is moved.

When a Point or section continuity mark is moved, it carries its associated knots with it, and when a knot associated with an SCM is moved, the opposite knot is rotated to keep the continuity of slope condition true.

A section may be split into two sections with an SCM between them with the menu option "Split", then selection of the section to be split.

A section may be converted to a straight line by clicking "Straighten", then selecting a section; the knots are moved to the one third and two thirds positions on the line.

Clicking the option "Postscript" causes a request for a file name; the suffix `.ps` is added and the current set of curves at the current scale is written the file for printing by a PostScript device.

The option "Redraw" draws the current set of curves with small squares for the Points and the knots shown; whereas "Drawcurves" draws just the curves.

The option "Write_bdy" causes a request for a file name; the suffix `.bdy` is added and the current set of curves is written to the file for ingestion by `meshtool` and DIME.

The option "Explicit" means that when in Adjust mode, a Point, knot or SCM is selected with the mouse as usual, but the new position is given by typing coordinates rather than clicking with the mouse. Clicking "Explicit" again toggles back to interactive mode.

In the "EXTRAS" submenu are options for naming Points and Curves for later use by a DIME application (see Section 10.5), and also options for creating a grid of specified size on the screen, and for creating a snap interval for rounding mouse clicks to the nearest point of the snap grid.

There is no 'undo' feature in curvetool: for detailed work it is best to start a script file right at the beginning, so that in case of problems, you may quit the program, remove the last few lines of the script file, and run the script to get back to just before the error.

WARNING: If Curves intersect each other, the Curves will no longer define a domain, and `meshtool` will not be able to make a triangulation.

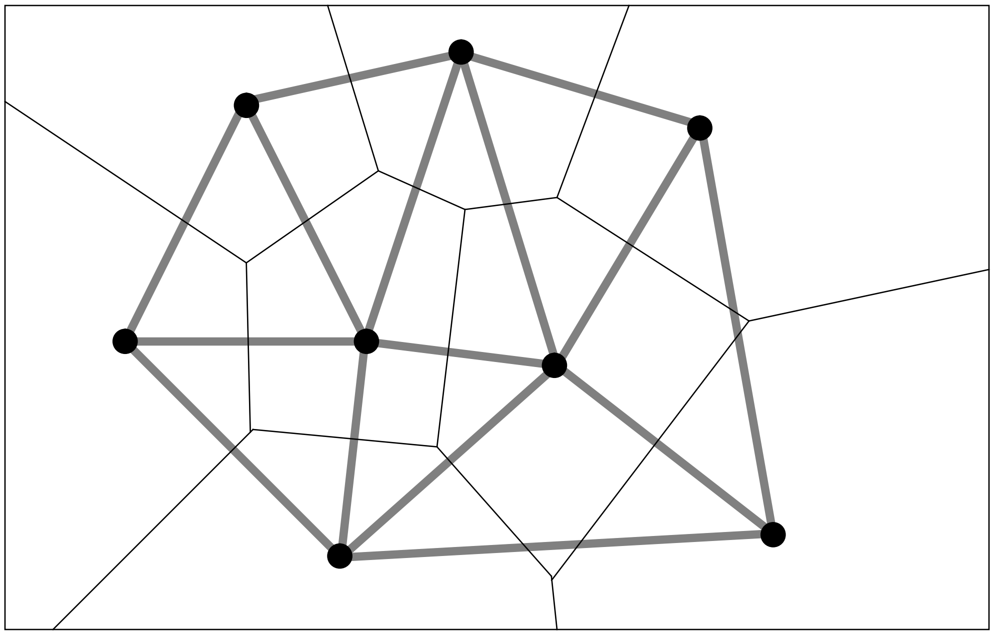
7.0 Meshtool

The program `meshtool` is for taking a boundary definition and creating a triangulation of certain regions of it. The graphical interface has the same look as a DIME program.

7.1 Meshtool Theory

The program `meshtool` adds nodes to an existing triangulation using the Delaunay triangulation[2]. A new node may be added anywhere except at the position of an existing node. Figure 7 illustrates how the Delaunay triangulation (thick gray lines) is derived from the Voronoi tesselation (thin black lines).

Figure 7 Voronoi Tesselation and Resulting Delaunay Triangulation



Each node (shown by a blob in the Figure) has a "territory", or Voronoi polygon, which is the part of the plane closer to the node than to any other node. The divisions between these territories are shown as thin lines in the Figure, and are the perpendicular bisectors of the lines between the nodes. This procedure tesselates the plane into a set of disjoint polygons and is called the Voronoi tesselation.

The Delaunay triangulation may be derived by connecting those nodes whose territories have a common border. This procedure does not always produce a unique triangulation of a set of nodes, the ambiguity arising from the question of whether territories that meet at a corner are touching or not. Part of the ambiguity may be resolved by demanding that only triangles of non-zero area be produced. Given this, we take any of the available triangulations. For example, four nodes at the corners of a square may be triangulated with the diagonal running either way, and `meshtool` may produce either one arbitrarily.

The Delaunay triangulation merely triangulates a set of nodes with no reference to the boundary structure of the domain we wish to mesh. We would like to classify the nodes as being associated with the Points or Curves of the domain, or neither; and also to classify the edges of the mesh as being associated with the Curves of the domain or not[12].

When a new node is added to the triangulation, it carries a label about whether it lies on a Point or Curve or neither. If it lies on a Curve it carries the arc-length measured along the Curve from its start Point to the position of the node. Each node that lies on a Curve has a pointer to the next and previous node (in the sense of arc length) which is on that Curve. We may determine if the topology of the mesh is equivalent to that of the given domain by asking if each node is mesh-connected to the next and previous nodes, and if not to place a new node on the Curve whose arc length is the mean of the two who should be connected.

This is the function of the menu option “Triangulate” in `meshtool`: to check the equivalence of the boundary and the mesh topologies, and if necessary add new nodes until they are equivalent. The green lines in the `meshtool` display show where mesh edges should occur, black shows actual mesh edges, and magenta shows the desired boundary.

In addition to the topological equivalence, there is another condition that the mesh should satisfy in order that DIME’s refinement algorithm not fail. This is discussed in the next Section, and requires a certain amount of user input.

The practical consequence of all this is that the actual boundary, in magenta, should run close to the mesh boundary, in green. If not, new nodes should be added on the boundary near the discrepancy.

7.2 Meshtool in Practice

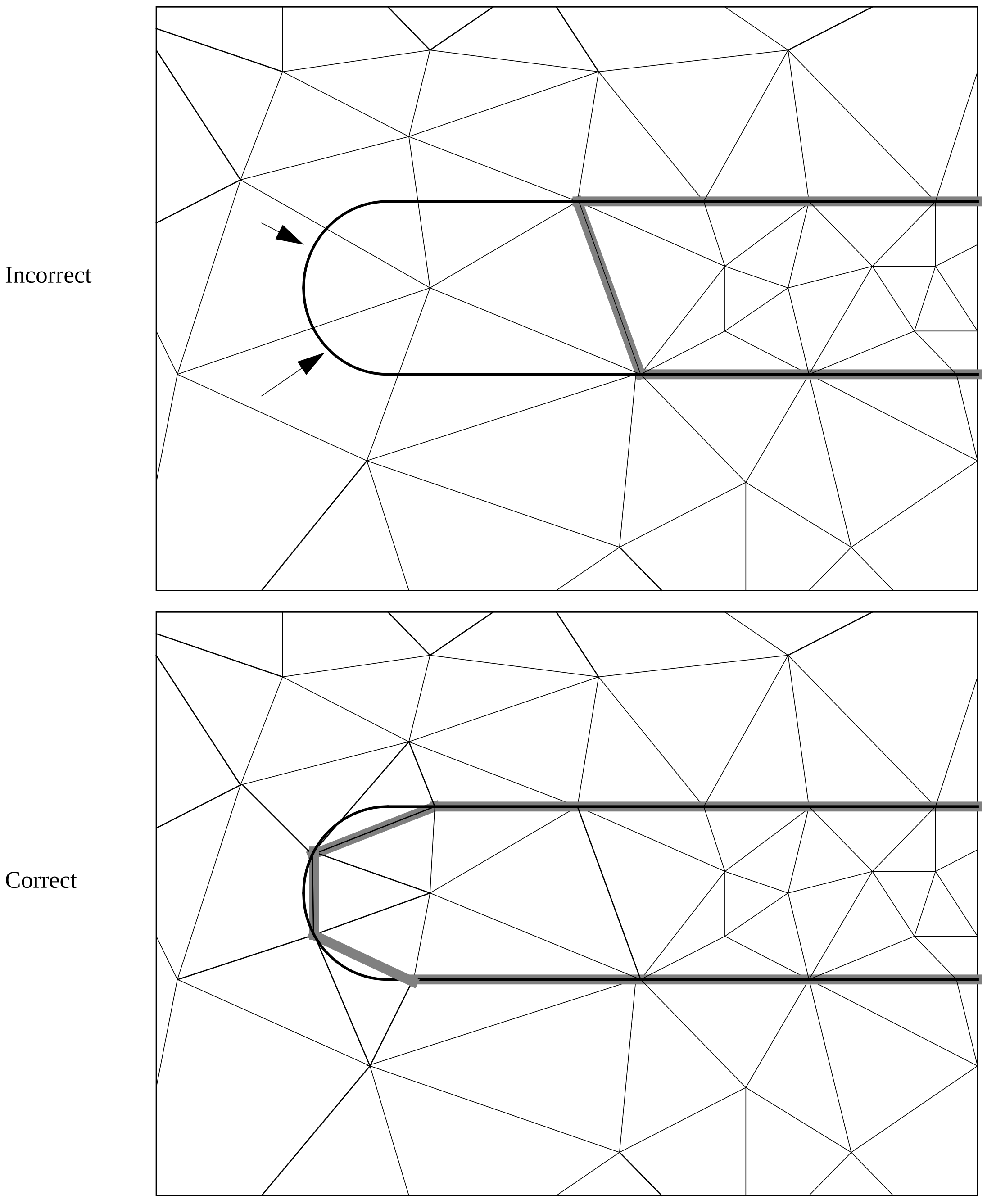
As soon as meshtool has started, a boundary file should be read in, by clicking “Readbdy” and supplying a file name.

`meshtool` operates by inserting nodes at the Points of a boundary description, on the Curves, or other places. The first thing it does is to put in a node at each Point of the boundary, and at the midpoint of each Curve. Thus the initial display is the boundary itself plus a perhaps confusing collection of black, green and magenta lines.

Click on the menu option “Triangulate” to create the minimal triangulation of the given boundary description, such that the topology of the mesh is equivalent to the topology of the boundary description. There may be places such as at the top of Figure 8 where the mesh does not seem to be quite correct.

The boundary description shown with a heavy line, does not satisfactorily match the mesh boundary, shown with a gray line. The solution is to put extra nodes into the mesh, perhaps at the positions shown by the arrows. The result is shown at the bottom of Figure 8, where the mesh boundary now closely matches the model boundary. If such corrections are not made at this stage, then refinement of the mesh will fail during the execution of the DIME program. More precisely, it is sufficient that the mesh boundary should match the model boundary in the following sense; if the model boundary is completely contained in the elements which share a mesh edge.

Figure 8 Topologically Correct But Insufficient Mesh, with addition of new nodes



Clicking the menu option “Addpoints” causes a blue stripe to appear over the menu area. When the stripe is present, mouse clicks are interpreted as new nodes to be added to the mesh, until the click is in the stripe, at which point the menu is active again.

Similarly, clicking the option “Edgenodes” creates a magenta stripe; further mouse clicks are interpreted as nodes to be added on the model boundary. The node is added at the closest point to the mouse position which lies on the model boundary.

After adding nodes, it is a good idea to click “Triangulate” again, which checks that the topology of the mesh is equivalent to the boundary topology. If necessary new nodes are added to make it so.

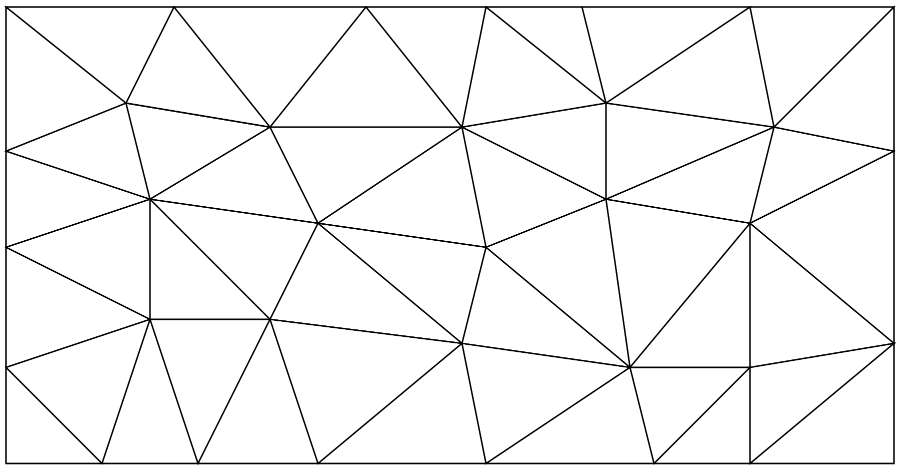
When the triangulation is complete, a mesh file may be created with the “Writemesh” option. You will be asked for a file name, and the suffix `.mesh` is automatically added and the file opened. You are prompted to click the mouse somewhere inside the region you wish to mesh. If all goes well, green triangles fill the required region, the mesh is written, and the file closed.

8.0 Mesh Structure

8.1 Sequential Structure

In Figure 9 is shown a triangular mesh covering a rectangle, and in Figure 10 the logical structure of that mesh on a single processor. The logical mesh shows the elements as shaded triangles and nodes as blobs. Each element is connected to exactly three nodes, and each node is connected to one or more elements. If a node is at a boundary, it has a boundary structure attached, together with a pointer to the next node clockwise around the boundary.

Figure 9 A mesh covering a rectangle



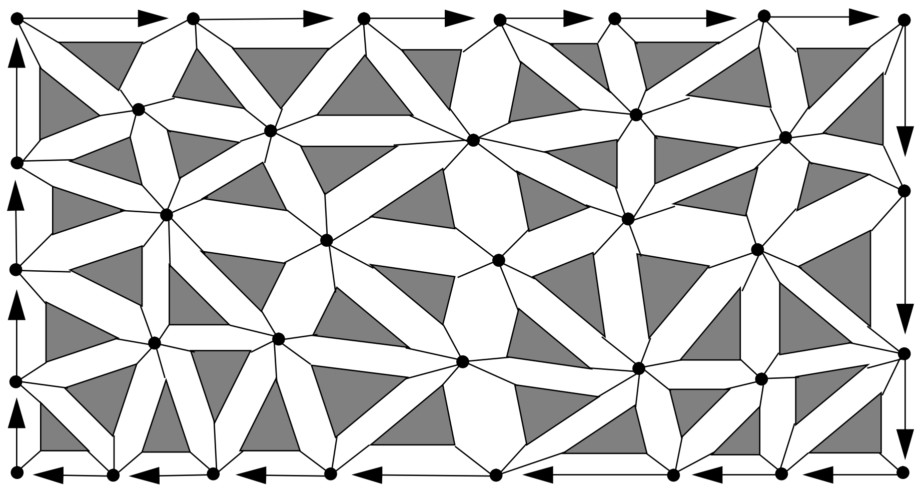
Each node, element and boundary structure has user data attached to it, which is automatically transferred to another processor if load-balancing causes the node or element to be moved to another processor. DIME knows only the size of the user data structures. Thus these structures may not contain pointers, since when that data is moved to another processor the pointers will be meaningless.

Just as there is a distinction between nodes and elements for the two-dimensional mesh, so there is a conceptual difference between boundary nodes and boundary elements for the one-dimensional boundary. However one dimension is a special case: for each boundary node there is a boundary edge between it and the next node clockwise; and for each boundary edge there is a node which is the one immediately anticlockwise of the edge. Thus DIME needs only one boundary structure, which may be used for boundary node data or boundary edge data.

8.2 Parallel Structure

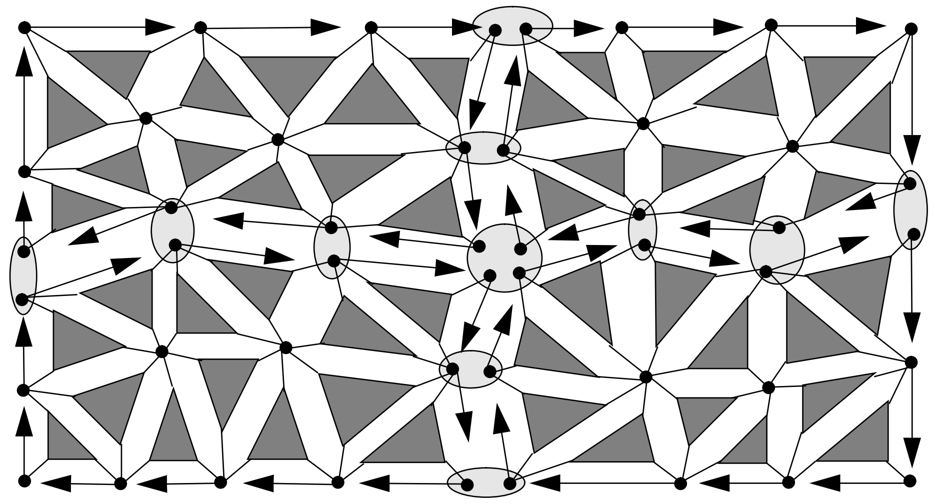
Figure 11 shows the structure of the mesh when it is split among four processors. Each element is owned by a processor and still has three neighbors, but some of the nodes have been split. The shaded ovals in the Figure are *Physical Nodes*, each of which consists of one or more *Logical Nodes*. Each logical node has a set of *aliases*, which are the other logical nodes

Figure 10 The Logical Structure of the Mesh of Figure 9



belonging to the same physical node. The physical node is a conceptual object, and is unaffected by parallelism; the logical node is a copy of the data in the physical node, so that each processor which owns a part of that physical node may access the data as if it had the whole node.

Figure 11 The Logical structure of the Mesh Split Among Four Processors



DIME is meant to make distributed processing of an unstructured mesh almost as easy as sequential programming. However, there is a remaining ‘‘kernel of parallelism’’ which the user must bear in mind. Suppose each node of the mesh gathers data from its local environment (i.e. the neighboring elements); if that node is split among several processors, it will only gather the data from those elements which lie in the same processor and consequently each node will only have part of the result. We need to combine the partial results from the logical nodes and return the combined result to each. This facility is provided by a macro in DIME called

`NODE_COMBINE`, which should be called each time the node data is changed according to its local environment. See Section 10.3 for more details.

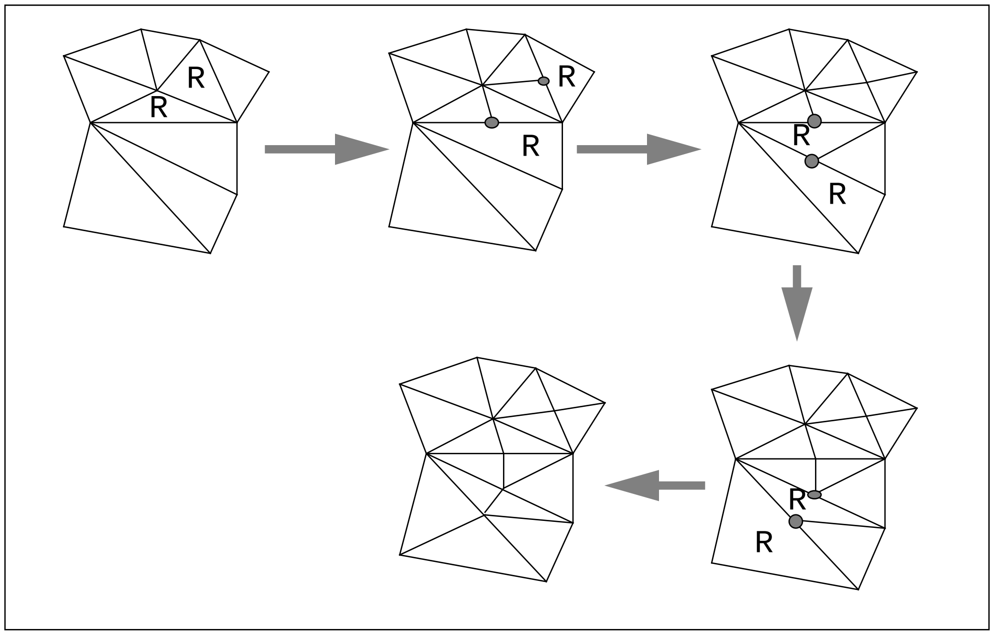
9.0 Refinement, Relaxing and Topological Relaxing

9.1 Rivara Refinement

The Delaunay triangulation[2] used by `meshtool` would be an ideal way to refine the working mesh, as well as making a coarse mesh for initial download. Unfortunately, adding a new node to an existing Delaunay triangulation may have global consequences; it is not possible to predict in advance how much of the current mesh should be replaced to accommodate the new node. Doing this in parallel requires an enormous amount of communication to make sure that the processors do not tread on each others toes[13].

DIME uses the algorithm of Rivara[14] for refinement of the mesh, which is well suited to loosely synchronous parallel operation. Figure 12 illustrates the process. First a number of elements are nominated for refinement, marked by a letter R in the top left of the Figure. For each of these elements, a new node is placed at the midpoint of the longest edge of the element.

Figure 12 Rivara Refinement



The mesh is no longer triangular though: on the other side of the edge with the new node is an element with four nodes, one of which is T-shaped or *non-conforming* (marked in the Figure with a blob). Now each of these elements with non-conforming nodes is labelled for refinement, and again a new node is placed at the midpoint of the longest edge.

The process is continued until there are no more non-conforming nodes. When an element is refined, there may be no need to actually create a new node if the element on the other side of the longest side has already created one. Notice how only two elements were initially nominated for refinement, which led to eight being refined in order to maintain the integrity of the mesh.

Whenever a new node is created, the user has a chance to interpolate user data from the neighborhood to be placed into the new node and elements. In addition, the decision about which is the longest side depends on a distance metric, which may be user controlled by providing a function `sqdist`, which returns the square of the distance between two nodes.

9.2 Topological Relaxation

The Delaunay triangulation has certain desirable properties when the mesh is to be used as a Finite Element mesh. In particular, the Laplacian operator for linear elements is diagonally dominant, resulting in considerable speedup of some iterative solvers[15].

Once a Delaunay triangulation has been refined by the Rivara method, it is no longer a Delaunay triangulation of the nodes. The process of topological relaxation changes the connectivity of the mesh to make it a Delaunay triangulation.

Each non-boundary edge of the mesh has a triangle on each side, and is thus the diagonal of a quadrilateral. If the sum of the two angles opposite the diagonal is greater than 180 degrees, then the diagonal is flipped to the other possible position in the quadrilateral. Since the sum of all four angles of the quadrilateral is 360 degrees, this diagonal will not be flipped back.

Topological relaxation consists of examining each non-boundary edge, and flipping it if necessary until no more flips can be made. The user may provide a callback function `swops_func` to interpolate element data whenever this flip occurs.

9.3 Relaxation

It is usually desirable to avoid triangles in the mesh which have particularly acute angles, and topological relaxation will reduce this tendency. Another method to do this is by moving the nodes toward the average position of their neighboring nodes; a physical analogy would be to think of the edges of the mesh as damped springs and allowing the nodes to move under the action of the springs.

The nodes are treated differently depending on whether they lie on Points or Curves of the domain boundary, or neither. The nodes at Points do not move. Nodes that lie on neither Points nor Curves move half the distance to the average position of their neighboring nodes. Nodes that lie on Curves move to a point corresponding to the average arc length of the nodes before and after them on the Curve, so that they slide along the Curve.

The nodes are relaxed in an arbitrary order, so that the result depends on the splitting of the mesh between processors. Also, it may be desirable to relax the mesh several times, to try to approach the state where each node is actually at the average position of its neighbors.

10.0 Programming

The program that you write to make a DIME application is to be run in each processor of a parallel machine: the same program runs in each processor, but with different pieces of mesh under the control of the different processors.

A DIME program consists of a special function `user_main`, which is called as soon as DIME has set itself up, and some other functions which take no arguments and return `int`. In addition there must be three structures defined, which describe the data the user wishes to identify with each node, element and boundary of the mesh. Each of these structures must have a size at least eight bytes. The minimum DIME program is

```
#include “dime.h”
struct user_node {char unused[8];};
struct user_bdy {char unused[8];};
struct user_elmt {char unused[8];};

user_main(user_menu, argc, argv)
MNUPTR user_menu;
int argc;
char **argv;
{
   DIME_PREAMBLE;
}
```

When `user_main` is called, a menu stub is passed, plus any program arguments not recognized by DIME. The macro DIME_PREAMBLE is necessary to pass to DIME the sizes of the user structures; the size is all that DIME knows about the user structures and is necessary so that it can allocate space for these whenever new nodes and elements are created.

The user structures SHOULD NOT CONTAIN POINTERS, since when a structure is moved from processor to processor, these pointers would be meaningless.

The program above, when compiled and linked to the DIME library, is sufficient to read in a boundary and mesh file (previously created with `meshtool`), to decompose the mesh among several processors, to refine, relax and topologically relax the mesh, and to draw various pictures of the mesh.

10.1 Menus

Suppose we wish to add two functions which count the number of nodes and elements respectively. We would replace the code in `user_main` with:

```
int ecount(), ncount();
DIME_PREAMBLE;
menu_add_function(user_menu, “Count_elmts”, ecount);
menu_add_function(user_menu, “Count_nodes”, ncount);
```

which appends the strings ‘‘Count_elmts’’ and ‘‘Count_nodes’’ to the menu labelled ‘‘USER’’, and associates the relevant function with each. We will show the code for these functions in the next Section.

We may also add a new menu subtree like this,

```
MNUPTR freddy;
int jimmy();
freddy = menu_add_menu(user_menu, “FREDDY”);
menu_add_function(freddy, “Jimmy”, jimmy);
```

so that the label FREDDY appears in the user menu, and when this is selected, the single option “Jimmy” appears, which would call the function `jimmy` if selected. In addition the option “TO_ROOT” is available in all the submenus.

WARNING: Labels for menu options should not contain spaces; use an underbar instead.

10.2 Accessing the Mesh

The functions `ecount()` and `ncount()` could be coded as:

```
ecount()
{
   ELMTPTR elmt;/* Pointer to element */
   int nelmt = 0;
   FORALLELMTS(elmt)
      nelmt++;
   NEXTELMT(elmt)
   fmulti(stdout);
   printf(“Proc %d has %d elmts\n”, procnum, nelmt);
   fsingl(stdout);
}

ncount()
{
   NODEPTR node;/* Pointer to node */
   int nnode = 0;
   FORALLNODES(node)
      nnode++;
   NEXTNODE(node)
   fmulti(stdout);
   printf(“Proc %d has %d nodes\n”, procnum, nnode);
   fsingl(stdout);
}
```

The macros `FORALLELMTS ... NEXTELMT` and `FORALLNODES ... NEXTNODE` are loops over all the elements or nodes owned by a processor, if there are any. The printing of the result is done in multiple mode because each processor in general has a different number of elements and nodes. The external variable procnum contains the processor number. Running this program sequentially simply prints out the result for one processor, whose processor number is zero. This is because a sequential machine is just a parallel machine with one processor. Note the return to the default single mode after using multiple mode.

The user may wish to access the coordinates of a node, using `node->x` and `node->y` for the horizontal and vertical coordinates. Other useful parts of the node structure are a pointer to its boundary structure, `node->b`, which is NULL if the node is not on a boundary; and the pointer to its user structure, which is `node->user`.

The boundary structure has a pointer to the next node in each direction. Moving along the boundary with the mesh on the right leads to the node `node->b->clock`, and the opposite direction leads to the node `node->b->antik`. The boundary structure also has an integer `node->b->type`, being one of the macros `POINT`, `CURVE` or `INSIDE`, if the node is associated with a Point, Curve or neither respectively. If the node is on a Curve, then the `double node->b->s` gives the approximate arc length along the curve to this node. Also, of course, there is a pointer to the boundary user structure `node->b->user`.

The element has a user structure `elmt->user`, and three pointers to its neighboring nodes, referred to as `elmt->neigh[i]` where `i` is 0, 1 or 2,which are in anticlockwise order around the element. A useful construction is `elmt->neigh[(i+1)%3]`, which is the next node anticlockwise after `elmt->neigh[i]`.

10.3 Parallelism

Suppose we wish each node to have in its user structure the sum of the areas of the elements around it. We could declare the user structures as

```
struct user_elmt {
   double earea;
};
struct user_node {
   double narea;
};
```

where `earea` is the area of the element, and `narea` is the node area - the desired sum of areas of neighboring elements. We can easily set up the element areas with one of the DIME functions (`area`):

```
double area();
FORALLELMTS(elmt)
   elmt->user->earea = area(elmt);
NEXTELMT(area)
```

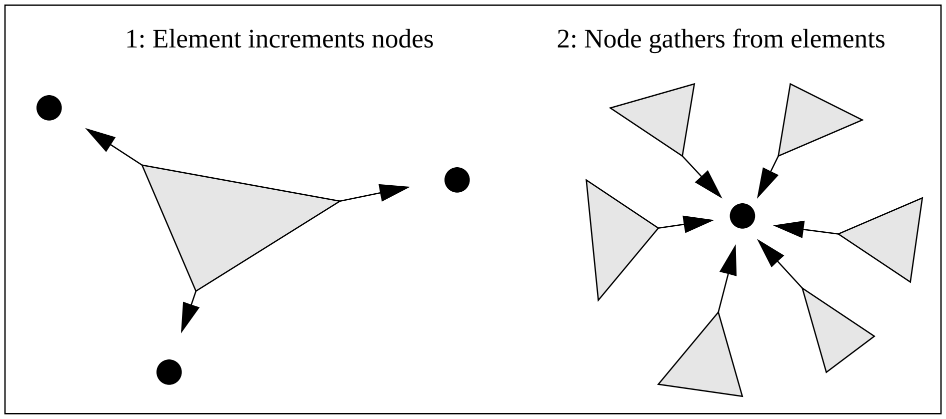
which illustrates how to access the user data. We may compute the node area in two ways, either by having each element give its area to its three neighboring nodes, or by having each node get the area from its neighboring elements:

```
/* Nodal Area: Method 1 */
FORALLNODES(node)
   node->user->narea = 0;
NEXTNODE(node)

FORALLELMTS(elmt)
   for(i=0; i<3; i++){
      node = elmt->neigh[i];
      node->user->narea += elmt->user->earea;
   }
NEXTELMT(elmt)
```

Which demonstrates how an element can access the data of its neighboring nodes. The second method is:

```
/* Nodal area: Method 2 */
FORALLNODES(node)
   node->user->narea = 0;
   FORALLNEIGH(node, elmt)
      node->user->narea += elmt->user->earea;
   NEXTNEIGH(node, elmt)
NEXTNODE(node)
```

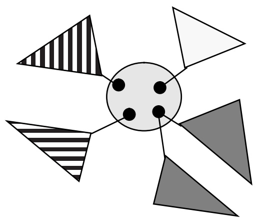
Which demonstrates the macro `FORALLNEIGH ... NEXTNEIGH` for looping over all the elements surrounding a given node. These two code segments may be visualized as follows:



With either method, each node now has the sum of areas of its surrounding elements. If the mesh is distributed, some nodes will only have a partial sum, and we need to combine these partial sums:

```
NODE_COMBINE(narea, fsum, 1);
```

For each physical node, the specified part of the user structure is summed over the logical nodes which constitute that physical node. For nodes which are not split, of course nothing happens. After the return from `NODE_COMBINE`, each logical node has the same value as its aliases, being the desired sum of areas:



Before NODE_COMBINE	Logical nodes have 2, 1, 4, 2
After NODE_COMBINE	Logical nodes have 9, 9, 9, 9

The combining function `fsum` is for summing real numbers (doubles); there are other combining functions provided - see the manual pages for details. The combining function must be a commutative and associative binary operation.

10.4 Global Data

The macro `NODE_COMBINE` allows each node of the mesh to access the data in its immediate neighborhood. Often we need more global information. Suppose we would like to know the total number of elements in the mesh, rather than the totals from each processor, as in Section 10.2. We calculate the value for each processor as before with a loop over all elements, but add the call:

```
GLOBAL_COMBINE(nelmt, fsum, 1);
```

which on return has replaced the quantity under the pointer (`nelmt`) with the sum of that quantity over all processors. The result may now be printed:

```
printf(“Total elements %d\n”, nelmt);
```

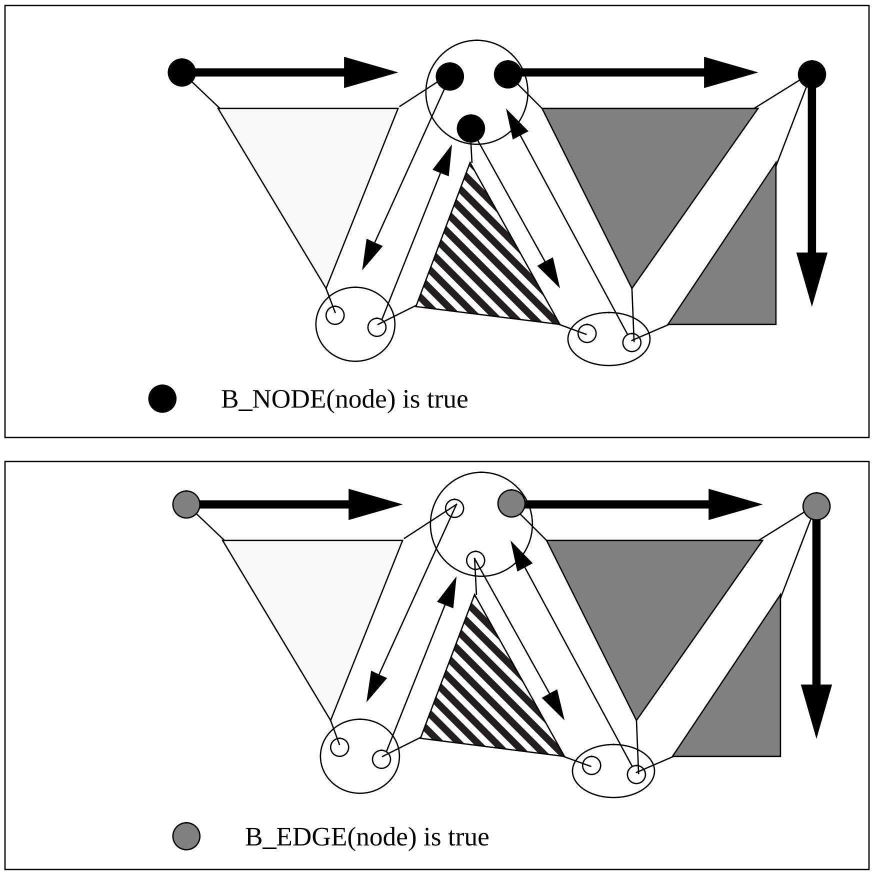
without recourse to multiple mode.

10.5 Boundaries

Nodes which lie either on a physical boundary of the domain or on a boundary between processor domains have a non-NULL value for their boundary structure, `node->b`. If this is true, there will be a boundary user structure associated with that node, and those data may be associated with either the node itself, or the edge connecting the node to the next one clockwise. We need to find out whether the node is on the physical boundary or just a processor boundary, and similarly for the edge. There are two macros for this.

Figure 13 shows part of the logical structure of a mesh near the boundary with three processors shown by different shading styles:In the top panel is shown the use of the macro `B_NODE(node)`, which is true (1) if and only if the node lies on a physical boundary rather than merely a processor boundary. The macro `B_EDGE(node)` is similar but subtly different. It returns true only if the edge clockwise of the node lies on the physical boundary.

Figure 13 B_NODE and B_EDGE macros



We would also like to know whether the boundary node lies at a Point or a Curve of the physical boundary and which one; and if so which Curve the clockwise edge might lie on.

The macro `B_POINT(node)` returns a pointer to the Point of the boundary specification with which the node is associated, or NULL if the node is not associated with a Point.

The macro `B_CURVE(node)` returns a pointer to the Curve of the boundary specification with which the node is associated, or NULL if the node is not associated with a Curve.

The macro `B_EDGE_CURVE(node)` returns a pointer to the Curve of the boundary specification with which the edge clockwise from the node is associated, or NULL if that edge is not identified with a Curve.

Analogous to `NODE_COMBINE` described in Section 10.3, there is a `BDY_COMBINE`, which can be called with the name of a member of the user boundary structure. This is then combined across the members of all their physical nodes which lie on the physical boundary.

10.6 Callback Functions

There are a small number of *callback functions* which the user may wish to set. These are used by DIME at various times without being explicitly selected from the menu. Examples of their use may be found in the manual pages.

- `distort` is a mapping from the coordinates `node->x` and `node->y` of a node to its position on the screen. This is the identity mapping by default.
- `sqdist` defines the metric of the two-dimensional manifold being meshed. It returns the square of the distance between two nodes, and is used by the refinement procedure to decide the longest side of a triangle.
- `refine_func` is called whenever an element is refined by placing a new node at the midpoint of its longest side. The user may fill the user structure of the new node with data interpolated from the neighborhood.
- `swops_func` is called when topological relaxation occurs. The user may fill the user structures of the two new elements with data interpolated from the old elements and surrounding nodes.
- `user_read` is called when a node, boundary or element structure is read from a file. The user may read data saved from a previous run or added with a text editor.
- `user_write` is called when a node, boundary or element structure is written to a file. The user may write data to be read by a future run of the program. This is useful as a backup facility during long batch runs.

10.7 Graphics

DIME provides graphics functions for drawing a mesh, drawing a logical mesh, or for contouring a function defined by linear finite elements at the nodes. The user may also use any functions from the Plotix parallel graphics library[9] which comes with Express.

10.7.1 Contouring

The contourer is sent a function pointer, and the function should take a node pointer as input and return a double. This value is the quantity that is contoured, either with lines or with color-coded shading, with explicit minimum and maximum contour values or by finding the actual minimum and maximum of the function. For more details see the manual pages.

10.7.2 PostScript hard copy

There is also a facility for making PostScript[10] hard copy. If the “Postscript” item is chosen from the DIME menu, you are prompted for a file name. The suffix `.ps` is added and the file opened if possible. From now on all the graphics which goes to the graphics part of the screen is also put into this file. A second selection of the “Postscript” item switches this off and closes the file. This file may then be printed on any PostScript device. At the top of the PostScript file are defined two variables “frame” and “color”:

```
/frame false def
/color true def
```

which control whether the PostScript is to be used as input to a Framemaker document or not, and whether the printing device can print in color. Either of these may be changed with a text editor before printing.

10.7.3 Zooming

The submenu ZOOM contains the four choices. “Extents” zooms to the smallest rectangle enclosing the whole mesh, and “Window” prompts for the corners of a rectangle and zooms so that the rectangle fills the screen. The option “Half” doubles the scale so that everything is half the size it was, and “Explicit” prompts for numerical values for the window size.

10.7.4 Drawing the Mesh

The mesh may be drawn by selecting “Drawmesh” from the DIME menu or by calling the function `drawmesh()`, and the logical mesh (similar to Figure 11) by selecting “Logicalmesh” or by calling the function `logicalmesh()`. For more details see the manual pages.

10.7.5 Flushing Graphics

If graphics is produced using the Plotix functions such as `move`, `cont`, `initpanel`, `panelpoint`, `endpanel`, `color`, etc, or the DIME extensions `draw`, `draw_elmt`, `arrow`, then the graphics stream must be loosely synchronously flushed with the Plotix call `usendplot()`. See the Plotix documentation and Sections 4.2 and 4.3 for more details.

11.0 References

1. R. D. Williams, *DIME: A programming Environment for unstructured triangular meshes on a distributed memory parallel processor*, Proc. 3rd Hypercube Conference, Pasadena, CA, 1988, ed. G. C. Fox.
2. A. Bowyer, *Computing Dirichlet Tesselations*, Comp. J. **24** (1981) 162.
3. R. D. Williams, *Supersonic Flow in Parallel with an Unstructured Mesh*, Concurrency, Practice and Experience, **1** (1989) 51.
4. R. D. Williams, *Performance of a Distributed Unstructured-Mesh Code for Transonic Flow*, Caltech Concurrent Computation Report C3P-856 (January 1990).
5. R. D. Williams, *Distributed Irregular Finite Elements*, J. Num. Meth. Fluid Mech. (to be published), also Caltech Concurrent Computation Project Report C3P 715.
6. C. F. Baillie, D. A. Johnston and R. D. Williams, *Computational Aspects of Simulating Dynamically Triangulated Random Surfaces*, Comput. Phys. Commun., (to be published).
7. R. D. Williams, B. Rasnow and C. Assad, *Hypercube Simulation of Electric Fish Potentials*, Proc. 5th Distrib. Mem. Computing Conf., Charleston SC April 1990.
8. R. D. Williams, *Free-Lagramge hydrodynamics with a distributed-memory parallel processor*, Parallel Computing **7** (1988) 439.
9. **Express:** *An Operating System for Parallel Computers*;
   Cubix: *Programming Parallel Computers without Programming Hosts*;
   Plotix: *A Graphical System for Parallel Computers*;
   ParaSoft Corp., 2500 E. Foothill, Pasadena, CA 91107, (818) 792 9941.
10. Adobe Systems Inc., *PostScript Tuorial and Cookbook; PostScript Language Reference Manual*; Addison-Wesley, Reading, MA, 1987.
11. G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon and D. W. Walker, *Solving Problems on Concurrent Processors*, Prentice-Hall, Englewood Cliffs NJ 1988.
12. W. J. Schroeder and M. S. Shephard, *Geometry-Based Fully Automatic Mesh Generation and the Delaunay Triangulation*, Int. J. Num. Meth. Engng. **26** (1988) 2503.
13. E. W. Felten and R. D. Williams, *Distributed Processing of an Irregular Tetrahedral Mesh*, Caltech Concurrent Computation Project C3P-793 (May 1989).
14. M-C. Rivara, *Design and Data Structure of Fully Adaptive Multigrid, Finite-Element Software*, ACM Trans. in Math. Software, **10** (1984) 242.
15. D. M. Young, *Iterative Solution of Large Linear Systems*, Academic Press, New York, 1971.

DIME Menu Tree

`USER`	As defined by User(§ 10.1)
`DIME`	See Below
`SCRIPT`	See Below
`ZOOM`	See Below
`Terse`	Toggles verbose mode for graphics
`Pause`	Waits for a newline from keyboard: useful for pausing a script file
`Erase`	Erases the screen leaving the menu only
`Quit`	Quit

`Readmesh`	Asks for boundary and mesh file and reads the mesh (§ 3.1, 4.4)
`Writemesh`	Asks for mesh file name and writes the mesh (§ 4.4)
`Drawmesh`	Draws the mesh
`Logicalmesh`	Draws the logical structure of the mesh (§ 8.1, 8.2)
`Postscript`	Toggles PostScript recording of graphics (§10.7.2)
`Rectrefine`	Refines a chosen rectangle so all edges shorter than chosen resolution (§ 3.4)
`Relax`	Moves each node to the average position of its neighbors (§ 9.2)
`Toporelax`	Topological relaxation (§ 9.3)
`Balance`	Load balancing by orthogonal recursive bisection (Manual Pages)
`Memory`	Summary of memory usage

`Use_script`	Requests and uses a script file (§3.8)
`Make_script`	Requests a file name and starts recording menu choices (§ 3.8)
`End_script`	Stops recording menu choices

`Extents`	Zooms to the smallest rectangle enclosing the mesh (§10.7.3)
`Window`	Requests rectangle and zooms to chosen window (§10.7.3)
`Half`	Halves the scale (§10.7.3)
`Explicit`	Requests position of window origin and width of window (§10.7.3)

Meshtool and Curvetool Menu Tree

Meshtool (Section 7)

`ZOOM`	As above
`SCRIPT`	As above
`Readbdy`	Requests and reads a boundary definition file (§5.1, §7.2)
`Redraw`	Erases and redraws current boundary and triangulation
`Addnodes`	Treats subsequent mouse clicks as nodes to be added to the triangulation (§7.2)
`Edgenodes`	Treats subsequent mouse clicks as nodes to be added at nearest boundary point (§7.2)
`Triangulate`	Adds enough nodes to make triangulation topologically equivalent to boundary (§7.1)
`Writemesh`	Requests point in region and writes mesh file
`Erase`	Erases the screen leaving the menu
`Quit`	Quit

Curvetool (Section 6)

`ZOOM`	As above
`SCRIPT`	As above
`EXTRAS`	See below
`Redraw`	Draws current curves with decoration
`Addpoint`	Next mouse click is position of new Point
`Connect`	Select two Points to be connected
`Adjust`	Allows Points, knots and SCM's to be moved
`Split`	Splits a cubic spline into two
`Straighten`	Changes a cubic spline to a straight line
`Drawcurves`	Draws current curves without decoration
`Postscript`	Prompts for file and makes PostScript file of current curves
`Write_bdy`	Prompts for file and makes `.bdy` file for meshtool
`Erase`	Erases the screen leaving the menu
`Quit`	Quit

`Grid`	Toggles grid on and off. If on, prompts for grid spacing
`Snap`	Toggles snap on and off. If on, prompts for snap spacing
`Name_point`	Select a point and name it
`Name_Curve`	Select a Curve and name it

DIME structures

The following is a simplified list of the DIME structures with a tiny explanation for each:

```
typedef struct node {                          Node Structure
   double x, y;                                Position of node
   struct bdy *b;                              Boundary structure or NULL
   struct nodeneigh *n;                        Pointer to list of neighbors
   struct user_node *user;                     User Data
} _NODE, *NODEPTR;

typedef struct bdy {                           Boundary Structure
   struct node *clock;                         Next node clockwise
   struct node *antik;                         Next node anticlockwise
   int type;                                   POINT, CURVE or neither
   union {
      struct curve *curve;                     Pointer to Curve
      struct point *point;                     Pointer to Point
   } bdy;
   double s;                                   Arc length if node is a Curve
   struct user_bdy *user;                      User Data
} _BDY, *BDYPTR;

typedef struct nodeneigh {                     List of elements neighboring a node
   struct elmt *elmt;                          Pointer to this elmt
   struct nodeneigh *next;                     Next anticlockwise in neighbor list
} NODENEIGH, *NODENEIGHPTR;

typedef struct elmt {                          Element Structure
   struct node *neigh[3];                      3 neighbor nodes of this elmt
   struct user_elmt *user;                     User Data
} _ELMT, *ELMTPTR;

typedef struct point {                         Point Structure
   char name[20];                              Name of the Point
   int ptnum;                                  Number of the Point
   double x, y;                                Position of the Point
} _POINT, *POINTPTR;

typedef struct curve {                         Curve Structure
   char name[20];                              Name of the Curve
   int curvenum;                               Number of the Curve
   double length;                              Length of the Curve
   struct point *first, *last;                 Points at the start and end of the Curve
   struct segment *start;                      List of Curve segments
} _CURVE, *CURVEPTR;
```

NAME

balance, set_balance, balance_orb

Load balancing a mesh

SYNOPSIS

```
void balance()

void set_balance(elmt, new_processor)
ELMTPTR elmt;
int new_processor;

void balance_orb()
```

FUNCTION TYPE

`balance, balance_orb -` Loosely Synchronous
`set_balance -` Local

DESCRIPTION

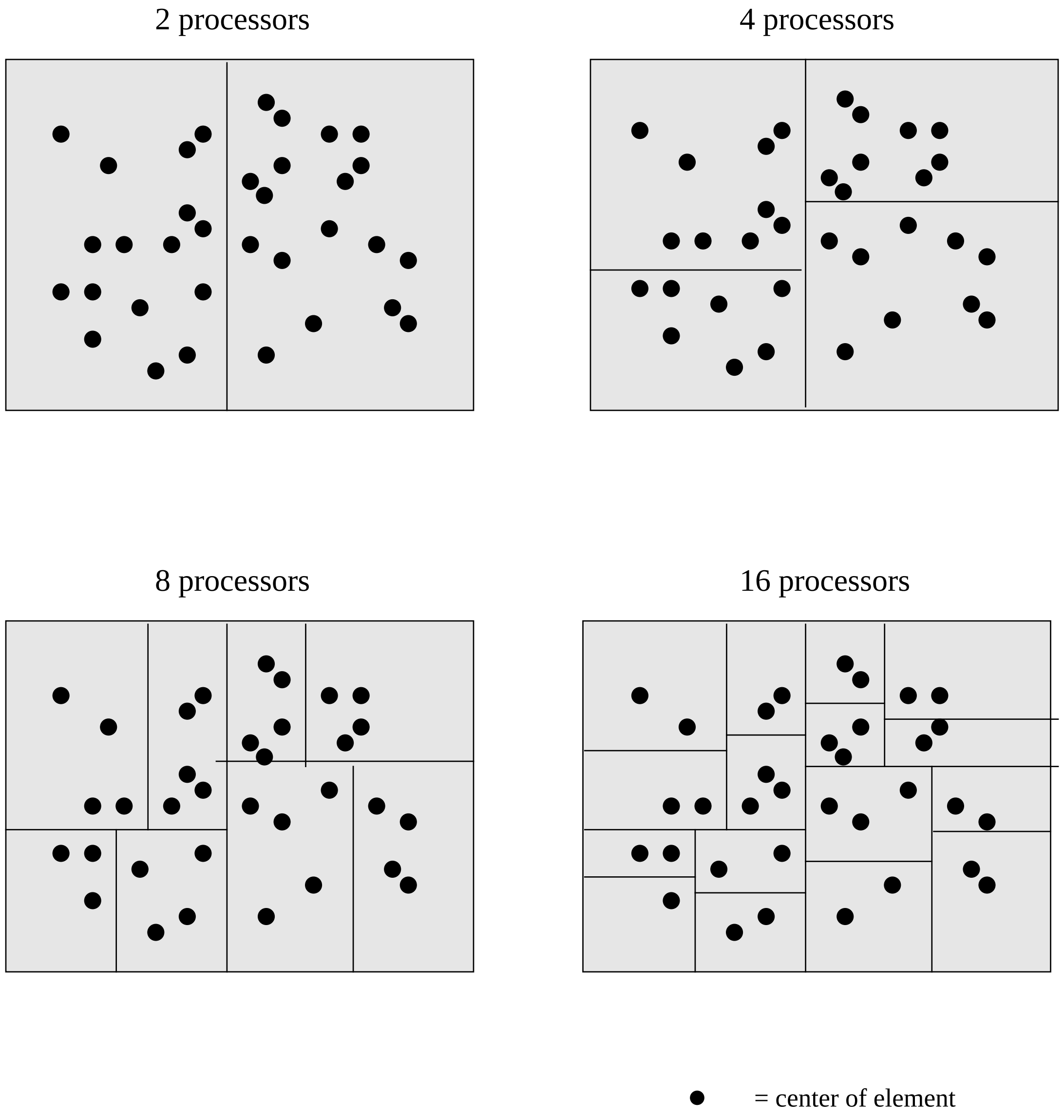
`balance` causes the mesh to be redistributed among the processors of the machine. Before calling `balance`, which must be loosely synchronous, the function `set_balance` should be called for each element. For example:

```
FORALLELMTS(elmt)
   set_balance(elmt, 6);
NEXTELMT(elmt)
balance();
```

would cause the entire mesh to be put into processor 6, leaving all the other processors idle. This is not a good load-balancing strategy.

/over

`balance_orb()` causes the mesh to be load-balanced by orthogonal recursive bisection. The mesh is split according to the position of the centers of the elements. First a line of constant x coordinate is found which splits the mesh into equal numbers of elements, then for each half a line of constant y is found which splits the half into quarters, and so on alternating x and y:



NAME

refine, refine_fraction, set_refine, refine_func

Refine a selection of mesh elements

SYNOPSIS

```
void refine()

void refine_fraction(func, fraction)
double (*func)(), fraction;

void set_refine(elmt, to_be_refined)
ELMTPTR elmt;
int to_be_refined;

void refine_func(elmt, node, node1, node2, new, elmt1,
elmt2)
ELMTPTR elmt, elmt1, elmt2;
NODEPTR node, node1, node2, new;
```

FUNCTION TYPE

`refine, refine_fraction -` Loosely Synchronous
`set_refine -` Local
`refine_func -` Callback

DESCRIPTION

Causes the mesh to be refined by the algorithm of Rivara. Before calling refine, which must be loosely synchronous, the function `set_refine` should be called for each element. For example:

```
#include “dime.h”
FORALLELMTS(elmt)
   set_refine(elmt, TRUE);
NEXTELMT(elmt)
refine();
```

would cause all the elements to be refined.

`refine_fraction()` causes a fraction `fraction`, between 0 and 1, of the elements of the mesh to be refined. The function `func` should be declared

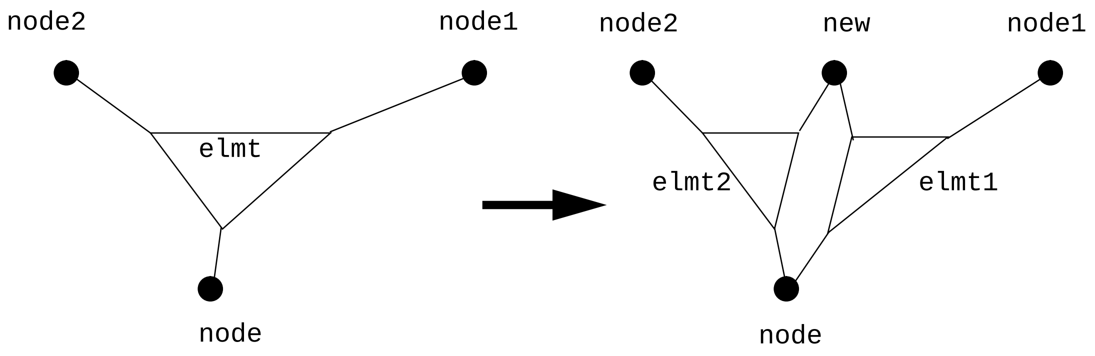
```
double func(elmt)
ELMTPTR elmt;
```

This function is called for each element of the mesh, and the value returned should be a measure of how worthy is that element for refinement. A value is found by binary search such that the given fraction of the elements are to be refined, then the refinement occurs.

Interpolation of the user's data may occur if the callback function `refine_func` has been set in the `user_main` function of the user code. The user's callback function is called after the new structures have been attached to the existing mesh and the old structures removed. The old structures are deleted immediately after return from the user's callback function. An example of the use of this callback is:

```
#include “dime.h”
user_main()
{
   void my_refine_func();
   refine_func = my_refine_func;
   ...
}

void
my_refine_func(elmt, node, node1, node2, new, elmt1, elmt2)
ELMTPTR elmt, elmt1, elmt2;
NODEPTR node, node1, node2, new;
{
   new->user->datum =
      0.5*(node1->user->datum + node2->user->datum);
}
```

which would cause the user's value `datum` to be linearly interpolated from the nodes around it. The meaning of the arguments is shown below, on the left being the old view and on the right the new view. The element `elmt` will be deleted when the function returns.



NAME

swops, swops_func

Topological Relaxation

SYNOPSIS

```
user_main(...)
{
   void my_swops_func();
   swops_func = my_swops_func;
}

swops()

void my_swops_func(A, B, C, D, ABC, ACD, ABD, BCD)
NODEPTR A, B, C, D;
ELMTPTR ABC, ACD, ABD, BCD;
```

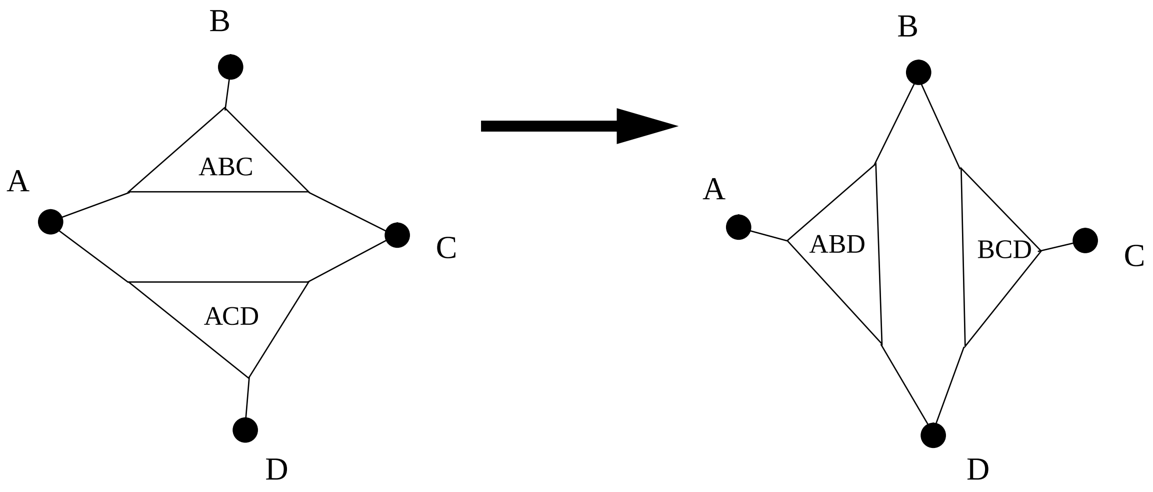
FUNCTION TYPE

`swops` - Loosely Synchronous
`swops_func` - Callback Function

DESCRIPTION

The function swops causes topological relaxation of the mesh, as explained in Section 9.3.

If `swops_func` is set, it is called each time an edge is swopped to allow the user to reset data in the user structures accordingly. Data from the old elements ABC and ACD should be interpolated to the new elements ABD and BCD, perhaps using user data from the nodes A, B, C and D. See the manual page for `refine_func` for setting up this callback.



NAME

distort, sqdist

Non-Euclidean metric

SYNOPSIS

```
user_main(...)
{
   void my_distort();
   double my_sqdist();
   distort = my_distort;
   sqdist = my_sqdist;
}

void my_distort(node, ax, ay)
NODEPTR node;
double *ax, *ay;

double my_sqdist(node1, node2)
NODEPTR node1, node2;
```

FUNCTION TYPE

Callback Functions

DESCRIPTION

If `distort` has been set, it is called whenever the coordinates of a node are required, for example in drawing the mesh, or deciding if a node is inside a user-selected window. `distort` takes a pointer to a node and two pointers to `double`, which should be filled with the screen-coordinates of the node.

If sqdist has been set, it replaces the usual Eulidean distance measure $(x0-x1)^2 + (y0-y1)^2$ with a user-supplied metric. The function is used during refinement to decide which is the longest side of a triangle.

NAME

menu_add_function, menu_add_menu

add functions and submenus to menu tree

SYNOPSIS

```
void menu_add_function(menu, label, function)
MNUPTR menu;
char *label;
int (*function)();

MNUPTR menu_add_menu(menu, label)
MNUPTR menu;
char *label;
```

FUNCTION TYPE

Loosely Synchronous

DESCRIPTION

In the function `user_main`, the user has the opportunity to add functions and submenus to the available menu. This is the only way that the user-provided functions can be executed by DIME. For example:

```
user_main(user_menu, argc, argv)
MNUPTR user_menu;
int argc;
char **argv;
{
   int doit(), freddy();
   MNUPTR submenu;
   menu_add_function(user_menu, “Doit”, doit);
   submenu = menu_add_menu(user_menu, “SUBMENU”);
   menu_add_function(submenu, “Freddy”, freddy);
```

would make the user menu have two items, `Doit` and `SUBMENU`. If `Doit` is clicked or in a script file, then the user-provided function `doit` is executed, and if `SUBMENU` is clicked, the item `Freddy` appears, which if chosen would execute the user-provided function `freddy`. Each user-provided function is called with no arguments.

NAME

draw, draw_elmt, arrow, disp_erase, color

Graphics functions

SYNOPSIS

```
int draw(node1, node2)
NODEPTR node1, node2;

int draw_elmt(elmt, shade)
ELMTPTR elmt
int shade;

int arrow(x, y, dx, dy)
double x, y, dx, dy;

int disp_erase()

int color(shade)
int shade;
```

FUNCTION TYPE

Local

DESCRIPTION

`draw` draws a line from node1 to node2 in the current line-drawing color.

`draw_elmt` draws a triangle a little smaller than an element in color `shade`.

`arrow` draws an arrow from (x, y) to (x+dx, y+dy) in the current line drawing color.

`disp_erase` erases the graphics part of the screen and redraws the current menu.

`color` changes the current line drawing color to the color `shade`. See `$DIME/src/include/color.h` for an explanation of the color map. See the Plotix documentation for an explanation of color in general.

When using these and the Plotix functions in parallel, the graphics must be loosely synchronously flushed with `usendplot()`. See Section 10.7.5.

NAME

contor, slow_contor, elmt_contor

Contouring functions

SYNOPSIS

```
contor(nfunc, nlevel, panels, min, max)
double (*nfunc)(), min, max;
int nlevel, panels;

slow_contor(nfunc, nlevel, panels, min, max)
double (*nfunc)(), min, max;
int nlevel, panels;

   double nfunc(node)
   NODEPTR node;

elmt_contor(efunc, nlevel, min, max)
double (*efunc)(), min, max;
int nlevel;

   double efunc(elmt)
   ELMTPTR elmt;
```

FUNCTION TYPE

Loosely Synchronous

DESCRIPTION

`contor` and `slow_contour` draw a contour plot of a function whose values are defined at the nodes from the function `nfunc` and is linearly interploated elsewhere in the mesh domain.

`elmt_contor` fills the elements with color based on the value of the function `efunc`.

In each case the number of contour levels is `nlevel`, ranging from `min` to `max`. If `min` `max`, then the actual minimum and maximum of the functions `nfunc` or `efunc` is found and substituted. For `contor` and `slow_contor`, the plot is contour lines if `panels` is 0 `(FALSE)` and is a false-color shaded plot if panels is 1 `(TRUE)`.

`contor` attempts to minimise communication from the parallel machine to the graphics server by using large filled polygons, while `slow_contor` uses the more robust but slower method of contouring each triangle separately.

NAME

fmax, fmin, fsum, imax, imin, isum

ready made combining functions

SYNOPSIS

```
double global;
GLOBAL_COMBINE(global, fmax, n);

struct user_node {
   double user_struct_member;
   ...;
}
NODE_COMBINE(user_struct_member, fmax, n);
```

FUNCTION TYPE

Combining Functions

DESCRIPTION

Each of these functions may be used in the macro GLOBAL_COMBINE to get global data. For example:

```
do {
   ...
   error = 0;
   FORALLELMTS(elmt)
      if(elmt->user->error > error)
         error = elmt->user->error;
   NEXTELMT(elmt)
   combine(&error, fmax, sizeof(error), 1);
} while(error > tolerance);
```

This is a typical situation where an iterative process continues until some global maximum is sufficiently small. In the loop over elements, we calculate the maximum value of something in the element user-structure called `error`. The loop over elements calculates the maximum within each processor, which is not of course the same as the maximum over all processors. The `combine` call replaces the value(s) under the pointer by the maximum over all processors.

WARNING: If a decision is to be made about the future course of the program based on the magnitude of something like `error`, and it has not been `combine`'ed, then deadlock may occur.

When data is gathered into a node from its neighboring elements, there should be a NODE_COMBINE to spread the results over the logical members of a physical node. See Sections 10.3 and 10.4 for more details.

The other combining functions work in much the same way, being for maximum, minimum and sum of doubles and integers respectively.

A new combining function may be created as in the following example, in which we would like the minimum absolute value of a quantity:

```
fmodmin(a1, a2, size)
double *a1, *a2;
int size
{
   if(fabs(*a1) > fabs(*a2))
      *a1 = fabs(*a2);
   else
      *a1 = fabs(*a1);
   return 0;
}
```

NAME

get_double, get_gin, get_int, get_string

input from interactive user or script file

SYNOPSIS

```
void get_double(d)
double *d;

void get_gin(x, y)
double *x, *y;

void get_int(i)
int *i;

void get_string(s)
char *s;
```

FUNCTION TYPE

Loosely Synchronous

DESCRIPTION

These functions get input either from an interactive user of a DIME program, or from a script file. If the program is being run in interactive mode, with a script file being made, then whatever response comes from the user is also recorded in the script file. If the program is being run from a script file, the input is read from that script file. In interactive mode, `get_double`, `get_int`, and `get_string` expect the user to type in a double, integer or string respectively. `get_gin` expects the user to provide graphical input, ie to click the mouse at some point on the screen. For example:

```
double x, y;
printf(“Click the next position please\n”);
get_gin(&x, &y);
```

NAME

user_read, user_write

reading and writing user data to/from mesh files

SYNOPSIS

```
user_main(...)
{
   int my_read(), my_write();
   user_read = my_read;
   user_write = my_write;
}

my_read(str, type, ptr)
char *str;
int type;
char *ptr;

my_write(str, type, ptr)
char *str;
int type;
char *ptr;
```

FUNCTION TYPE

Callback functions

DESCRIPTION

These functions may be used to read and write data to and from the mesh file. If `user_write` is set in `user_main`, it is called when the mesh is written, first at the beginning for the output for “miscellaneous” data, then for each node, element and boundary that is written to the file. DIME first writes its own information about the structure, then sends the address of a character buffer of size 1024 bytes to the function pointer `user_write`, together with the integer `type,` specifying what structure is being written, and a pointer to that structure. If `user_write` returns 1 (TRUE), then whatever was written into the buffer is copied into the mesh file. If `user_write` returns 0 (FALSE), then nothing is added to the mesh file.

Similarly when a mesh is being read, if `user_read` has been set, the user may read in data from the mesh file to the user structures. In this case the return value is ignored.

Miscellaneous data is written and read in single mode, so all processors get the same data.

There should be no newline characters in the string written.

A mesh written with a user-defined write function may be read by a program without a user-defined read function: the data is simply ignored.

Here is an example of using these functions to keep some data with the mesh:

```
#include “dime.h”
char version[100];
struct user_node {...;};
struct user_elmt {...; int oblong; ...;};
struct user_bdy {...; double imgmar; ...;};

user_main(user_menu)
MNUPTR user_menu;
{
   int my_read(), my_write();
   user_read = my_read;
   user_write = my_write;

}

my_read(str, type, ptr)
char *str;
int type;
char *ptr;
{
   NODEPTR node;
   ELMTPTR elmt;
   BDYPTR bdy;
   switch(type){
      case MISC:
         sscanf(str, “%s”, version); break;
      case NODE: node = (NODEPTR)ptr;
         break;
      case ELMT: elmt = (ELMTPTR)ptr;
         sscanf(str, “%d”, &elmt->user->oblong); break;
      case BDY: bdy = (BDYPTR)bdy;
         sscanf(str, “%lf”, &bdy->user->ingmar); break;
   }
   return 1;
}
```

```
my_write(str, type, ptr)
char *str;
int type;
char *ptr;
{
   NODEPTR node;
   ELMTPTR elmt;
   BDYPTR bdy;
   switch(type){
      case MISC:
         sprintf(str, “%s”, version);
         return 1;
      case NODE: node = (NODEPTR)ptr;
         return 0;
      case ELMT: elmt = (ELMTPTR)ptr;
         sprintf(str, “%d”, elmt->user->oblong);
         return 1;
      case BDY: bdy = (BDYPTR)bdy;
         sprintf(str, “%lf”, bdy->user->ingmar);
         return 1;
   }
}
```