GridPro v5.5

Reference Manual for TIL

Program Development Corporation

300 Hamilton Ave., Suite 409

White Plains, NY 10601

Tel:(914)761-1732, Fax:(914)761-1735

Email:gridpro@gridpro.com

October 15, 2012

©**Copyright Program Development Corporation, 1992-2012 – Licensed Materials, All Rights Reserved**. This document contains proprietary and confidential information of PDC. The contents of this document may not be disclosed to third parties, copied, or duplicated in any form, in whole or in part, without the prior permission of PDC. The contents of this document are subject to change without notice and do not represent a warranty on the part of PDC.

Contents

Part I

Chapters

Chapter 1

Overview

A good general purpose grid (mesh) generator should be at least good on two accounts: 1) Quick and easy to setup typical complex gridding problems: The considerations include the first gridding turn-around time, the subsequent parametric design turn-around time, the modular parametric design (adding and subtracting features) turn-around time and the clustering capabilities (for CFD use); And 2) Good grid quality: such as the grid smoothness, orthogonality, desired grid distribution and surface fidelity. Both accounts are better served through automation with different levels of user selectable controls.

For a multi-block structured grid generator, automation can be classified into four areas: 1) Optimum distribution of high quality grid, 2) Book keeping of topological information, 3) Topology generation, and 4) Surface restructuring and repair.

To this end, GridPro is a general purpose, 3-dimensional, multi-block structured grid (mesh) generator using an advanced smoothing scheme that incorporates many automatic features.

1.1 What is GridPro?

GridPro has achieved full automation in high quality grid distribution and the book keeping of topological information. It partially automates topology generation by reducing the user task to the generation a coarse wireframe of the topology in which only imprecise corner and edge information is required; while the blocks and block faces are automatically generated from the wireframe. It also has a certain capability of automatic surface restructuring and repair, such as auto-stitching of surface gaps between surface patches, and implicit surface trimming and intersection capturing.

The design of GridPro has followed the principles: 1) Minimizing the user input with a strong emphasis on topological template (`COMPONENT`) construction capability and reusability, 2) Maximizing the grid quality, and 3) Optimizing the grid distribution.

The first principle cuts down both the initial setup time and more drastically the subsequent setup time for configuration modifications; The second translates into a higher solution accuracy, and faster convergence for the CFD solvers; And the third reduces the demand for computer resources in terms of both the CPU time and the amount of RAM usage.

Once the block topology is chosen, the process of grid generation using GridPro is accomplished by solving a variationally based system with an iterative updating scheme. In this process, the initial setup of the grid is only a guess to the final grid that is the converged solution of the system. Thus, in a general sense, the final grid (solution) is independent of the initial grid distribution. This results in that only imprecise initial position information is

required and dramatically reduces the amount of required user input while generating grids with excellent quality. On the other hand, multiple sweeps are needed to generate a grid. In this sense, it is more CPU intensive.

GridPro consists of two main modules:

Figure 1.1 shows the relationship among GridPro/`Ggrid` and its different components. The next two sections are devoted to the discussion of this relationship.

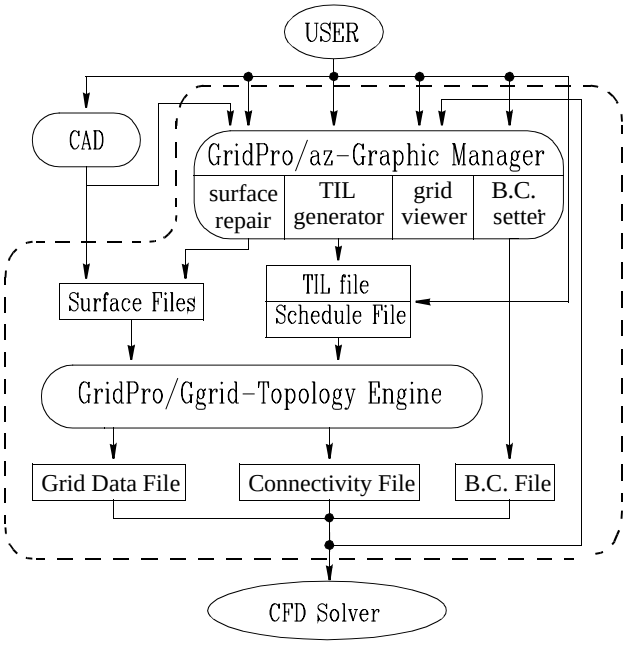


Figure 1.1: Relationship between GridPro/`Ggrid` and its environment.

1) `az`-Graphics Manager (type: '`az <ret>`' to start it).

2) `Ggrid` topology engine that reads in topology, and generates and writes out the grid. This part of the GridPro also includes a suite of other non-graphic utilities.

The media between the two modules is the Topology Input Language (TIL). The `az`-Graphics Manager is effectively a language generator that generates and feeds the TIL codes to `Ggrid` to generate grids. One can also manually write his/her own TIL codes and/or edit them without resort to the `az`-Graphics Manager.

An important point is that every TIL code is a template for the same class of problems.

This manual is about the non-graphic part of GridPro, which includes `Ggrid`, TIL and other GridPro utilities. Since the executable `Ggrid` is the center piece of this volume, we will use the terms GridPro and GridPro/`Ggrid` interchangeably, unless a real distinction is needed.

1.2 What do you put into `Ggrid` ?

For a run-case, the input required from a user has three components: surface specifications, a block topology and a run schedule.

The surface specifications constitute an external component in that they are provided mostly from the outside of the GridPro environment and conform to certain standards. The block topology is the static part of the grid generation process; Grid generation for a changed topology usually requires one to rerun GridPro. In contrast, the run schedule is the dynamic part of the

process. The run schedule should be designed to best use the computer resources and to guide the convergence. You can inspect the grid at the middle of the run and reschedule your run at any time you wish. You can also resume a previously stopped run.

In general, the grid generation process involves several iterations of modifying and enhancing the block topology to achieve the best grid quality and optimal grid distribution. The level to which the enhancements are brought is up to the user.

1.2.1 Surface specifications

Surface specifications are independent of the surface grid generated with GridPro and they can be from different sources and in different formats. The collection of implemented formats for GridPro is still expanding.

However, there are also some particular requirements on the surface conditions and formats. An important requirement is the smoothness of the surface defined. GridPro can handle up to 90^o jumps for the surface normal vector. Such condition can break down on the seam lines of many surfaces generated directly from popular CAD systems.

When necessary, surface geometries should to be restructured to conform to the requirements of GridPro before using them. Such restructuring can be as simple as changing the data format; or as complicated as merging surfaces, and smoothing, modifying and removing small features.

Within the GridPro software package, there are utilities and tools to assist users to create and restructure surfaces. The GridPro/az-Graphic Manager can also be used to accomplish these.

1.2.2 Block topology

The phrase *topology* here is defined as the connectivity information of block corners (not blocks!), the surface assignments of corners (and possibly edges and faces) and the initial positions of corners. A special topology input language (TIL) is used to record the topology into files. The topology files must have the file name extension '.fra'.

It is the user's responsibility to design and record a block topology. The design and recording process can be done either manually coding in TIL or using az-Graphic Manager. However, to record the design, a user does not need to provide the information about the edges, faces and blocks in the design. GridPro will generate such information automatically! In simple terms, a TIL file contains mainly a sequence of corner definitions, each of which provides an approximate initial position of the current corner, a list of other corners that has a link to the current corner, and a list of surfaces that the corner should be on.

An important feature of TIL is to organize the topology design into COMPONENTs. A COMPONENT works much the same way as a subroutine in, say, FORTRAN. It hides the irrelevant details of the topology and connects to the rest of the topology through interfacial variables. This feature of TIL provides a natural means to build reusable component libraries.

1.2.3 Run schedule

Since GridPro uses an advanced smoothing scheme in which the grid is generated in multiple sweeps just as in the case of the ordinary elliptic grid generation, a schedule for the run must be provided for a better and faster convergence.

A schedule file consists of step lines, each of which lists a sequence of actions that direct the run process of GridPro. The name of a schedule file must have the prefix part same as the corresponding main topology file and end with the file name extension '.sch'.

1.3 What do you get out from Ggrid ?

GridPro outputs the block connectivity information and grid data in various formats.

1.3.1 Block grid data

Block grid data is in simple point data format listed block by block.

1.3.2 Block connectivity data

Block connectivity data is written into the file 'conn.tmp'.

1.4 A complete and simple example

Our first example is to generate a grid for a simple 2-d case. The region to be gridded is the area between the circle and the rectangular box as shown in Figure 1.2(a). Through this

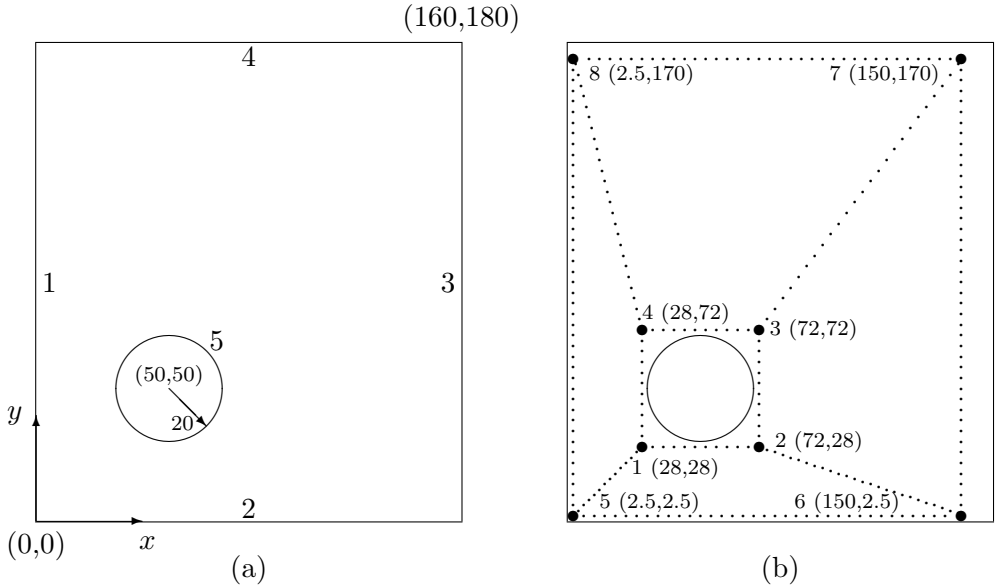


Figure 1.2: (a) A simple region to be gridded. (b) The block topology designed by a user.

example, we demonstrate what the basic steps are for generating a grid using GridPro. Also in this and other examples of this manual, we will focus on the details of coding in TIL; Therefore we will not use az-Graphic Manager to prepare the TIL code though it is much simpler to do so for simple cases like this. The advantage of manually coding in TIL becomes obvious when one deals with more complex cases with topologically repetitive structures, or design iterations are required.

1.4.0 Step 0 – Preparing surfaces

For the example we are going to demonstrate in this section, all the surfaces used are built-in implicit analytic surfaces which are simple. For this reason, the task of current step is significantly reduced.

Generally, surfaces need to be prepared to conform to the requirements of GridPro. For instance, GridPro requires a certain degree of smoothness for the surfaces. This sort of requirements and the surface data format requirements forms a tedious, but more or less independent part of GridPro; These, being general geometric items, are much less particular to GridPro. Therefore, a general discussion of this step is left to Chapter 7.

1.4.1 Step 1 – Partitioning and Labelling geometry

The first thing we do is to partition the geometry into surfaces and label the resulting surfaces. The word surface has a specific meaning for GridPro. A surface here is defined as a portion of the geometry on which the distribution of grid points is automatically created by GridPro. It is up to the user to decide how the geometry should be partitioned into different surfaces. Though there is no unique way to perform the division, for most cases there are one or two natural ways to do so. The way the geometry should be partitioned also strongly depends on the block topology in use and the existing geometric features. The surfaces should form a leakless closure of the gridding region. The gridding region must be connected.

For our case, we can consider that there are five surfaces in the problem. The circle is one, and the four sides of the box are the other four surfaces. The surface labels we assigned are marked in Figure 1.2(a) with 1, 2, 3, 4 and 5. In the assignment, each surface should have a unique number to identify it and the order of the numbering is not important. We will also use the phrase *surface id* to mean the number assigned to a surface.

In the above surface division, grid points intended for, say, surface 1 will not be distributed into surface 2. Another choice for the division is to regard surfaces 1, 2, 3 and 4 as one surface and assign it a single label. In this case, GridPro will consider the four sides of the box as a whole in order to decide the distribution of grid points on it.

So far, we have not said anything about the surface data formats and requirements. As said earlier, we leave the details to Chapter 7, except to mention here that as a rule, each of the surfaces should be relatively smooth. The intersections of different surfaces do not need to be explicitly defined; but, generally, the specifications of the surfaces should extend somewhat beyond the intersections. Note also that only the surface portion that is a part of the closure of the gridding region is really used. We will not have this problem here, however, since all the surfaces can be specified analytically in very simple terms as can be seen in the next step.

1.4.2 Step 2 – Designing a block topology with TIL

The second thing we do is to design a block topology for the region to be gridded. This can be done with the GridPro/az-Graphic Manager or simply using a pen on a piece of sketch paper. For the purpose of learning TIL, we will go the later route.

The process of designing the topology is the fun part of the whole grid generation process. It also needs some creativity.

At a simple level, the goal is to cover the region with quadrilaterals (for 2d cases). This covering does not need to be done at a geometric precise level. It needs to be done only at a rather topological level; That is, a surface can be represented as a set of piece-wise linear segments (again for 2d cases) placed not too far from the real surface. Let's take a circle as an example. It can be represented by a square, a pentagon, or an arbitrary polygon, all depending on the needs of your block design. The design will be programmed with the Topology Input Language (TIL) into a file to be compiled and processed by GridPro.

For a configuration of complex geometry, one can design simple components of block topologies, and assemble them into a complex one much the same way as a real airplane or

automobile is built.

For our case, the designed block topology is shown as the dotted lines and big dots in Figure 1.2(b). The solid lines are surfaces. The big dots represent the block corners and the dotted lines connecting the big dots are corner links. We also labelled the block corners from 1 to 8 with its approximate coordinates written next to it in the parentheses. The term *corner label* is also referred as *corner id*. The TIL program for this topology design is written in a file called 'example1.fra' and listed in Program 1.1.

Program 1.1 *File 'example1.fra'*

```
SET DIMENSION 2
SET GRIDDEN   16

COMPONENT circleInBox()
BEGIN
  s 1   -plane  ( 1.0    0  0     0) ; #x1 side
  s 2   -plane  (   0  1.0  0     0) ; #y1 side
  s 3   -plane  (-1.0    0  0 160.0) ; #x2 side
  s 4   -plane  (   0 -1.0  0 180.0) ; #y2 side
  s 5   -ellip  (0.05 0.05 0) -t 50.0 50 0 ; #circle

  c 1   28  28  0  -s 5 ;
  c 2   72  28  0  -s 5    -L 1 ;
  c 3   72  72  0  -s 5    -L 2 ;
  c 4   28  72  0  -s 5    -L 3 1 ;
  c 5   2.5 2.5 0  -s 1 2  -L 1 ;
  c 6   150 2.5 0  -s 2 3  -L 2 5 ;
  c 7   150 170 0  -s 3 4  -L 3 6 ;
  c 8   2.5 170 0  -s 4 1  -L 4 7 5 ;

  g 1 5  32;
END
```

To have an overview of a complete example, only a brief explanation is given here for Program 1.1. A detailed explanation is left to Chapter 2 to 4.

Program 1.1 tells GridPro that the topology is a 2d case (`SET DIMENSION 2`); Edges are initialized to have 16 grid points (`SET GRIDDEN 16`); And the topology consists of one component (`COMPONENT circleInBox()`) in which five surfaces and eight corners are defined. A corner is defined by its initial position, the surfaces it should be on (with the `-s` flag) and the corners it has link to (with the `-L` flag). Note: the coordinates must be specified in 3-d fashion for both 2d and 3d cases.

The line starting with the key word 'g' assigns the edge connecting corners 1 and 5 with 32 grid cells.

Any thing beyond the '#' character in a line is ignored by GridPro.

1.4.3 Step 3 – Scheduling your run

Our run schedule must be in the file 'example1.sch'.

To be simple, let's say we want to generate a grid in 100 sweeps and write out the 2d grid to a file called 'blk2d.tmp'. 'example1.sch' will have only two lines as follows:

Program 1.2 *File 'example1.sch'.*

```
step 1: -S 100 -w

write -f blk2d.tmp
```

The schedule section of the file has only one step with two actions. The actions are executed one by one from left to right. The first action '`-S 100`' tells GridPro to run 100 relaxation sweeps; The second, '`-w`' directs GridPro to execute an output grid action.

The details about the data to be outputted is specified in the output section of the same schedule file. This section consists of all the lines beginning with the key word `write`. For our case, it has also only one line, which tells GridPro to write out the grid to a file called 'blk2d.tmp' (`-f blk2d.tmp`).

1.4.4 Step 4 – Generating a grid

Now, we are almost ready to run GridPro. Since some '.tmp' files will be generated automatically in the current directory when running GridPro, it is always a good idea to create a directory for each run case, and place your '.fra', '.sch' and other relevant files in it. You should run GridPro in it too.

GridPro is normally installed in the directory '*SOMEWHERE*/GridPro'. Before running it, make sure the path has '*SOMEWHERE*/GridPro/bin' in it. You can check the path by typing,

```
set | grep path <ret>
```

If it is not set, set it by appending to '~/.cshrc' a line,

```
set path = ( $path SOMEWHERE/GridPro/bin )
```

To generate a grid, type,

```
Ggrid example1.fra<ret>
```

GridPro will read 'example1.fra' once to generate all topology information and schedule the run according to 'example1.sch'. When it finishes, the grid generated is stored in the file, 'blk2d.tmp'.

The data is in a simple point data format. That is, the data is listed block by block starting from block 1. For each block the data can be read from a FORTRAN program as follows:

Program 1.3 *Point data format in FORTRAN*

```
READ(UNIT,*) IMAX,JMAX,KMAX
DO 10 I=1,IMAX
DO 10 J=1,JMAX
DO 10 K=1,KMAX
10  READ(UNIT,*) X(I,J,K), Y(I,J,K), Z(I,J,K)
```

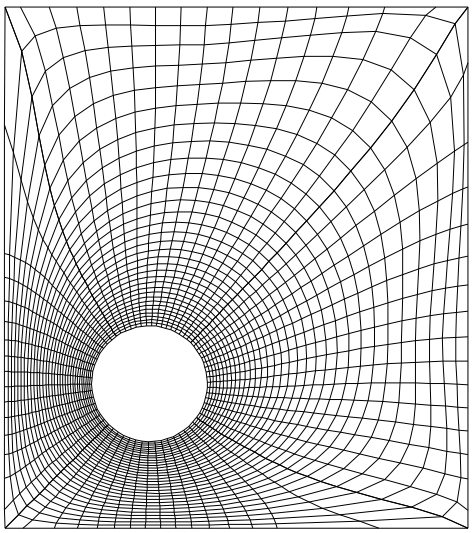


Figure 1.3: A grid generated from Program 1.1.

where `X, Y, Z` are the x, y, and z coordinates of a grid point.

The connectivity information of the blocks is automatically stored in the file 'blk2d.tmp.conn'. For the format of it please read Section 5.3.2.

Now you can examine the grid using the `az`-Graphic Manager by typing,

```
az -v blk2d.tmp <ret>
```

For the details of operating `az`, see the manual volume for GridPro GUI and the on-line help in `az`.

The grid is also shown in Figure 1.3.

So far, we have finished one cycle of the grid generation process. It is likely enough for a simple problem like this. However for a complex problem, the user probably has to go through several cycles of retopologizing and rescheduling to generate a grid to one's liking.

1.5 Organization of this manual

The rest of the manual is arranged as follows: In Chapter 2 to 4, the Topology Input Language (TIL) is introduced and explained through the simple example given in Chapter 1. The usage of vector and scalar expressions to specify real space positions is discussed in Chapter 5. How TIL program can open, close, read and write files is explained in Chapter 6. These are useful for parameterization of the topology. Chapter 7 is devoted to the surface specifications implemented in the current version of GridPro.

Appendix A and B are scheduling syntax and worked out examples.

Chapter 2

Basics of Topology Input Language (TIL)

Minimum or no knowledge of TIL is required for people using the az-Graphic Manager only to create topology. However, programming with TIL become increasingly important for complex geometries with repetitive sub-topologies, or when design optimization for the geometry is in consideration. In this Chapter, we will use Program 1.1 to illustrate the basics (the minimum knowledge required) of Topology Input Language (TIL).

2.1 TIL program structure

When GridPro processes a TIL program, anything from a '#' character to the end of the line is ignored. A '#' character can be used to introduce comments for that line.

New-line characters and tabs are treated as spaces and consecutive spaces will be truncated to a single space. Thus, the alignment in Program 1.1 is purely for styling purposes. In writing a TIL program, spaces can often be omitted as long as tokens can be read in correctly.

A TIL program can have three sections appearing in the following order:

1) An optional global assignment section.
2) An optional include section.
3) A component definition section.

Program 1.1 has only sections 1) and 3). The missing section 2) is normally used where the topology design is programmed in several files or topology libraries. For the case where the entire topology is contained in a single file, there should not be an include section.

The optional assignment section is used to assign certain global parameters. A parameter not assigned in the assignment section takes a default value.

The first two lines of Program 1.1 form the assignment section,

```
SET DIMENSION 2
SET GRIDDEN   16
```

It specifies the problem to be a 2d case and initializes every edge to have 16 grid cells. The parameter `DIMENSION` can have a value either 2 or 3. The parameter `GRIDDEN` must have a value greater or equal to 3. Without these assignments, `DIMENSION` and `GRIDDEN` take the default values 3 and 8, respectively.

The 'SET GRIDDEN' line can have another syntax as follows:

```
SET GRIDDEN  E_X_axis 16 E_Y_axis 12 CROSS 10
```

Multiple such 'SET GRIDDEN' lines can be used to initialize grid density on edges. Here, E_X_axis, E_Y_axis and CROSS are global edge labels defined in the TIL program, that each may be a collection of more than one edges. A 'SET GRIDDEN' line like this functions as a schedule step before those steps in the .sch file.

GridPro is a general purpose 3d software package. For a 2d case, GridPro will first convert it to a 3d case, then run it as a 3d problem. Thus, for a 2d case, anything in the topology file involving real space positions still has to be specified in a 3d fashion. However, the z coordinate should always be assigned a value, 0. The places where real space positions are involved can be the initial positions of corners, the data to specify surfaces, and possible translation and rotation operators for surfaces and components. We will see it when we go through the details of Program 1.1.

The component definition section is the core of a TIL program. The basic unit of topology specifications in TIL is a COMPONENT and a TIL program consists of at least one COMPONENT. A COMPONENT is composed by a set of declarations and statements bounded by the key words BEGIN and END as follows,

```
COMPONENT comp_name( arg_list )
BEGIN
   declaration
   .
   .
   declaration
   statement
   .
   .
   statement
END
```

Each declaration or statement starts with a key word and ends with the terminal symbol ';'. The first COMPONENT is always the head COMPONENT which does not require any arguments to be passed in and out. GridPro will construct the complete block topology from the head COMPONENT.

For our case, the entire topology design is specified in a single component named circleInBox. The five statements starting with an s define the five surfaces. The eight statements beginning with a key c define the eight topology corners shown in Figure 2.1(b). The last statement assigns a grid density for an edge.

2.2 Defining surfaces

A surface can be in one of three modes depending on how they are used. A surface of the fixed mode is fixed in space by the data specifying the surface; A surface of the periodic mode is used for periodic boundary conditions and is not fixed in space by the data specifying the surface. A surface of the float mode has no fixed position in the space. For most cases, surfaces

are defined in the fixed mode. A fixed surface can be either internal or external depending on whether both sides or only one side of the surface need to be gridded.

Before we proceed further, let us make a distinction between the phrases surface specification and surface definition used in this manual. By a surface specification, we mean the data and data format used to describe the shape of a surface. On the other hand, by a surface definition, we mean a statement in the TIL program which starts with a key word '`s`' and assigns an id and other attributes to a surface. The main function of a surface definition is to make the surface known to other parts of the TIL program.

2.2.1 Surfaces of the fixed mode

All the five surfaces used in Program 1.1 are in the fixed mode. They are also all used as external surfaces. The term external here simply means that the grid region is on one side of the surface, as opposed to an internal surface where blocks must appear on both sides of the surface.

External surfaces

Among the five surfaces, the first four are of type `-plane`; The fifth is of type `-ellip` (for ellipsoid).

There are two groups of surface types used in GridPro. The first group is the explicit surfaces. A surface of this group is usually specified by a fair amount of data stored in a separate file(s). The second group of types are implicit. In this case, a surface is defined as an equal potential surface of a scalar valued analytic function of position vector. Some of the simple forms of the functions are hard wired into GridPro. They are called built-in implicit types. For these types, a surface is specified by providing several parameters in the corresponding surface definition statement in the TIL program. There is no need for a separate specification data file. Both types, `-plane` and `-ellip` are built-in implicit types. (For a complete list of types accepted by GridPro see Chapter 7.)

For a surface of type `-plane`, the four real numbers enclosed in the parentheses (a b c d) specify the plane in such a way that a point on the plane satisfies the equation $ax+by+cz+d=0$ and the plane normal vector (a, b, c) should point into the region to be gridded. To be more specific, let us look at the statement defining surface 3,

```
s 3  -plane(-1.0  0  0  160);
```

This is the right side of the box (Figure 2.1(a)) with the normal vector pointing opposite to the x axis. Therefore the equation for it is $-x + 160 = 0$ and the parameters for the surface appear as `(-1.0  0  0  160)`.

For a surface of type `-ellip`, three real numbers enclosed in the parentheses (a b c) are provided. A point on the ellipsoid satisfies the equation $(ax)^2+(by)^2+(cz)^2-1=0$. For a circle of radius 20, we have $a = 0.05$, $b = 0.05$, and $c = 0$. The center of the circle is translated by the `-t` *translation_vector* operation with *translation_vector* = 50.0 50.0 0. Altogether this appears as,

```
s 5 -ellip(0.05 0.05 0) -t 50.0 50 0 ;
```

Note that the `-t` *translation_vector* operation can be used for any type of surface. In fact, general transformations can be applied to surfaces. And a general transformation can be specified using vector expressions (See Chapter 5).

As we mentioned above for plane surfaces, the surface normal must point into the region to be gridded. This is generally required for all types of external fixed surfaces. If the surface definition is in a wrong orientation, a `-o` flag can be placed in the corresponding surface definition statement to reverse the orientation. Normally, the place to put it is next to the type (and associated parameters or data) flag. For example, the following statement defines the same surface 3 as before,

```
s 3  -plane(1.0  0  0  -160)  -o;
```

Another useful attribute that can be associated with a surface is a targeted average off-wall normal grid spacing. The line,

```
s 3  -plane(1.0  0  0  -160)  -o -c 0.00001;
```

assigns surface 3 a spacing of 0.00001. It means that grid points near surface 3 are intended to be clustered toward surface 3 with a target first layer grid normal spacing 0.00001.

Two things you need to keep in mind. First, the clustering will not take effect until it is turned on in the run schedule. Turning it on or off can be scheduled at any step in the schedule file. Second, turning the clustering on does not always mean the targeted spacing will be reached. GridPro tries to limit the grid growth ratio of the length scales of two consecutive grid cells not larger than 3.

Internal surfaces

An internal surface is a surface for which both sides of the surface are to be gridded. An internal surface can be specified with any surface type that can be used for an external surface. However, the orientation of the surface must be suppressed with the flag **-O** in the surface definition statement. The grids on both sides of an internal surface will be matched on the surface. A typical example is the wake of flow over an airfoil. To obtain a high grid quality, it is desirable to have internal surfaces relatively flat.

2.2.2 Surfaces of the periodic mode

Though periodic boundary conditions are not used in the example programs, it is basic enough to render a discussion here.

A surface of the periodic mode can only be of implicit type (`-implic`, *-xpolar* or `-xyz`. The same `s` statement syntax defines the surface. However, the data used to specify the surface is different.

In this case, a surface specification really specifies a family of infinitely many surfaces and one of them will become the final surface determined by many other factors. For details see, Section 2.5 and 7.4.

2.2.3 Surfaces of the float mode

It can be used to specify spacings for block interfaces. The only valid surface type for this mode is `-float`.

2.3 Defining corners

The eight statements beginning with a key `c` in Program 1.1 define the eight topology corners shown in Figure 2.1(b). For each of the `c`-statements, the number next to the key `c` is the label we assigned to the corner; The next three numbers give an approximate initial position (x, y, z) of the corner. Two more pieces of information for defining a corner are the fixed boundary conditions of the corner and the linkages to other corners.

By a boundary condition, we simply mean a logical association or assignment of a corner, edge or face to a certain surface. GridPro uses the associations to determine the automatic distribution of grid points on corresponding surfaces.

The boundary conditions of the fixed mode for a corner are specified through a list of surface labels following `-s` and the linkages are specified by a list of corner labels following `-L`. All referenced corners or surfaces following the `-s` flag and `-L` flag should be defined before the current statement. For example, consider the statement,

```
c  6     150  2.5  0    -s  2  3    -L  2  5;
```

This says that corner 6 is on surface 2 and 3 in the final grid (`-s 2 3`), and has links to corner 2 and 5 (`-L 2 5`) . Notice, in Figure 2.1(b) corner 6 has also a link to corner 7, however by the rule, only those corners defined before corner 6 will be listed for corner 6. Thus the link from corner 6 to corner 7 will only appear in the link list for corner 7.

The initial position of corner 6 is at (150, 2.5, 0) which is not and does not need to be on surface 2 and 3. This is another point we would like to make: The surface assignments for a corner is meant for the final grid; the initial position of a corner does not have to be placed on the surfaces assigned to the corner. In fact, the initial positions of corners can be specified at a very imprecise level and, in theory, the final grid is independent of the initial positions. On the other hand, the initial positions can not be entirely arbitrary in order to have a convergent final grid.

A good rule of thumb is to put the links intended for a surface on the outside of the surface if it is a closed surface, and on the convex side of the surface if it is not, and more carefully place the corners that are near the high curvature region.

2.4 Assigning grid densities

Without an explicit assignment of grid density, an edge is assigned the default value of 8 cells. There are four means to change the assignment. They are listed with increasing priorities as follows,

1) Global assignment without edge labels: We have discussed it earlier in Section 2.1. An example is a line such as,

```
SET   GRIDDEN   3
```

in the header of the main TIL program file.

2) Local static assignment: It is done through the `g`–statements in the components of a TIL program. Assume we have the following line in component `circleInBox` of Program 1.1,

```
g   1 5 32    1 2 25    1 3 30    6 5 20    2 3 1;
```

Here each triplet of numbers following the key word, `g`, defines an edge grid density assignment. Let us use Edge(1,5) to mean the edge connecting corners 1 and 5. Thus, 1 5 32

means assigning 32 grid points to Edge(1,5). This assignment will propagate through all the parallel edges, that is Edge(2,6), Edge(3,7) and Edge(4,8) in this case.

Three rules apply here, a) All the corners referenced (i.e. corners 1, 2, 3, 5 and 6 in this case) must be defined before this statement. Otherwise, it is a syntax error. b) If an edge referenced does not exist(e.g. Edge(1,3) here) or the grid density is less than 3 (e.g. in 2 3 1), the triplet is ignored. c) If an edge group is affected more than once, the assignment with more grid points takes precedence (e.g. 1 2 25 takes precedence over 6 5 20). This last rule also applies over different `g`–statements in different components.

3) Global static assignments with edge labels: Act as if it is the first schedule step. Again, see Section 2.1 for an example.

4) Dynamic assignment: Grid density can also be dynamically changed in a similar fashion as for static assignments. It is done through `-g` actions in the schedule file. Changes can be scheduled to happen at different steps. For details, see Appendix A.

2.5 Periodic boundary conditions

`i` statements are used to define periodic boundary conditions. A periodic boundary is set by identifying corner pairs through a period with respect to a periodic surface.

Note that: A corner on a periodic surface should NOT be assigned to the surface through the '`-s`' flag in the corner definition statement!

Consider a simplified example of turbo blade cascades in 2d shown in Figure 2.1(a). Figure 2.1(b) is the topology design for one of the blades. The solid lines are surfaces in the fixed mode. The two dashed lines here are surfaces in the periodic mode. They both have the same surface label, 4, since they are periodic to each other with respect to surface 4.

Then, what do we mean by 'periodic with respect to a surface'? Let's explain it with the example in mind.

First, on the two dashed lines we must have the same number of corners and every corner on one dashed line is paired with one and only one corner on the other dashed line. The neighbouring relationship of corners on one dashed line must be preserved through the pairing process (that is, the two sides have the same topology). The pairing of two corners is also called an identification of the corners since the two corners are identical in the sense that a corner on the lower dashed line of blade 2 is the paired corner on the upper dashed line of blade 1. With a correct identification between corners on the two dashed lines, all the grid points on the two dashed lines are automatically identified (or paired). In our case, the identification is: corner 1 to corner 13, corner 2 to corner 14, corner 3 to corner 15, and corner 4 to corner 16.

Second, a surface specified in the periodic mode defines a coordinate transformation from the physical space (x, y, z) to some working space, say, (u, v, w), such that in the new coordinate system (u, v, w), two corners having an identifying relation have the same values for v and w, and a given and fixed difference δu in u.

Third, it is the user's responsibility to choose the coordinate transformation for which certain consistency must be maintained. In terms of the new system, any surfaces intersecting both dashed lines (surfaces 2 or 3) must be periodic in the same fashion. That is, in the (u, v, w) space and using the above example, if a point $(u0, v0, w0)$ is on surface 2 and near the lower-left intersection between surface 2 and surface 4 where the final grid is expected, the point $(u0 + \delta u, v0, w0)$ must be on the surface that intersects surface 4 on the upper left-intersection, which, in this case, happens to be the same surface 2.

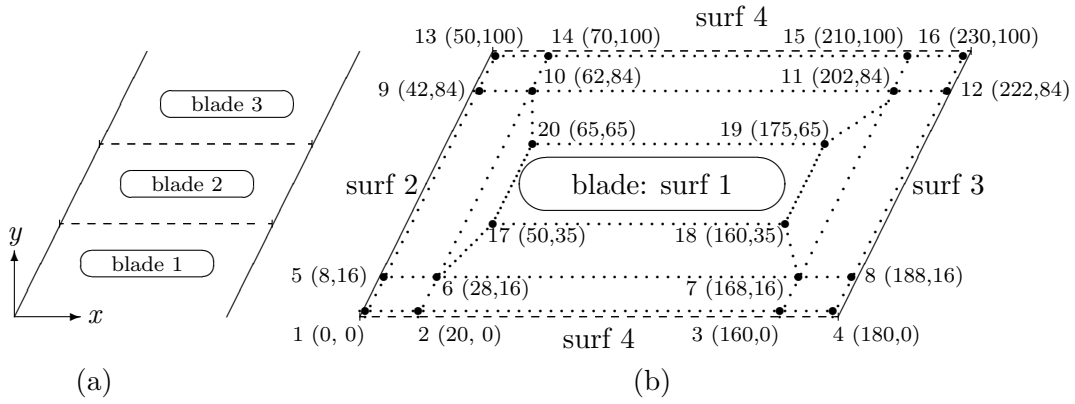


Figure 2.1: A turbo cascade in 2d.

In our case, the choice of coordinate transformation used for surface 4 is

$$u = x + 2y$$

$$v = 2x - y$$

$$w = z$$

Surface 2 and 3 are,

$$2x - y = 0 \qquad (\ v = 0\ )$$

and

$$-2x + y + 360 = 0 \qquad (\ -v + 360 = 0\ )$$

The TIL program for this topology is listed below,

```
COMPONENT blade()
BEGIN
  s 1 -linear "blade.dat";
  s 2 -plane ( 2 -1 0 0);
  s 3 -plane (-2  1 0 360);
  s 4 -implic "periodSurf.h" 250.0; # define the coordinate
                                    # transformation

  c 1    0   0 0 -s 2 ;
  c 2   20   0 0      -L  1;
  c 3  160   0 0      -L  2;
  c 4  180   0 0 -s 3 -L  3;

  c 5    8  16 0 -s 2 -L  1;
  c 6   28  16 0      -L  2  5;
  c 7  168  16 0      -L  3  6;
  c 8  188  16 0 -s 3 -L  4  7;

  c 9   42  84 0 -s 2 -L  5;
  c 10  62  84 0      -L  6  9;
  c 11 202  84 0      -L  7 10;
```

```
  c 12 222  84 0 -s 3 -L  8 11;

  c 13  50 100 0 -s 2 -L  9;
  c 14  70 100 0      -L 10 13;
  c 15 210 100 0      -L 11 14;
  c 16 230 100 0 -s 3 -L 12 15;

  c 17  50 35  0 -s 1 -L  6;
  c 18 100 35  0 -s 1 -L  7 17;
  c 19 175 65  0 -s 1 -L 11 18;
  c 20  65 65  0 -s 1 -L 10 19 17;

  i  (1 13  4);
  i  (2 14  4);
  i  (3 15  4);
  i  (4 16  4);
END
```

The last four statements set up the periodic boundary condition for the two dashed lines. An `i`-statement has the following syntax:

`i` (*cid1 cid2 sid*);

Here *cid1* and *cid2* are two corner ids, *sid* is a surface id. The statement identifies corner *cid2* with corner *cid1* through surface *sid*. Here, one has to make sure that surface *sid* is one that can be used for a periodic BC. In other words, it is one of the types `-xpolar`, `-xyz`, or `-implic`. And remember that not every `-implic` surface can be used for a periodic BC. More precisely, for the final grid in the new coordinate system (u, v, w) defined in surface *sid*, the periodic boundary condition requires,

$$(u_2, v_2, w_2) \text{ - } (u_1, v_1, w_1) = (period,0,0)$$

where (u_1, v_1, w_1) and (u_2, v_2, w_2) are for corners *cid1* and *cid2* respectively. Notice, the period, which is given when the surface is defined, is specified in terms of the u coordinate of the new system. The sign of *period* is irrelevant, since internally the sign is recalculated from the initial positions of relevant grid points. See Chapter 7, for the details of constructing an '.h' file for a periodic surface.

2.6 Topology building rules

In designing a block topology, one needs to keep in mind the automatic rules that GridPro uses in the topology construction, then, follow the user rules to build a valid topology.

2.6.1 Automatic rules

Automatic rules are rules that GridPro uses to build the final topology from the topology input program (TIL code).

1) *The rule of object building:* Unless explicitly overruled, a link forms an edge, a closed 4-edge loop forms a face (or a block for 2d cases) and six closed faces form a block.

This rule may generate false faces or blocks. GridPro provides a means to correct them either automatically or manually. In the case of Program 1.1, since the quadrilaterals defined by corners 1, 2, 3 and 4, and by corners 5, 6, 7 and 8 form two 4-edge loops, two blocks will be generated. However, they are not intended to be blocks. Therefore, they can be explicitly excluded with the `x`-statement such as,

```
x f  1 3  5 7 ;
```

where a quadrilateral is referenced by a pair of its diagonal corner labels. More discussion on statements like this appears in Chapter 4.

These loops will be also automatically excluded from forming blocks as long as the surfaces are all correctly assigned to corners.

2) *The rule of surface assignments:* Unless explicitly overruled, the surface assignments of an edge (face) are derived from the surface assignments of its two (four) boundary corners (edges) with the possible modifications by the next two rules.

Again, in Program 1.1, this rule sets the surface assignments for all edges. GridPro will determine that the edge connecting corners 7 and 8 is on surface 4 since both corners 7 and 8 are on surface 4; Further, since this edge is the only edge on surface 4, GridPro will automatically distribute the grid points of this edge onto the part of surface 4 bounded by the intersections of surface 4 with surface 1 and surface 3. By the same token, the edges defined by corners 1 and 2, corners 2 and 3, corners 3 and 4, and corners 4 and 1 are all on surface 5; GridPro will automatically distribute the grid points on all of these edges as a whole on the circle – surface 5.

3) *The rule of overlapping surfaces:* When a face is assigned either automatically or manually to two or more fixed surfaces, these surfaces are overlapping surfaces. The data specifying these overlapping surfaces must actually overlap each other over the area where the face may locate. In the part of topology that has surface overlaps, they are considered as one surface, thus, the intersections of the overlapping surfaces will not be sought.

Surface overlap is used for avoiding surface confusion for some tricky surfaces. In those cases, one has a pre-knowledge as to where a face of concern will or will not go in the final grid generated by GridPro, and by some reason, the face did not attach to the correct part of the surface. One can divide the surface of concern into several surfaces, and overlap them on the area where no confusion is likely. Note that, the concept of surface overlap here is not merely a physical space overlap of surfaces; It also attaches topological requirements. Two surfaces overlap only when at least one face is located on both of them. Note also, that overlap is only a local concept, in that, two surfaces may overlap in one region and intersect in another.

4) *The rule of reduction of surface assignments:* It is an over supply of surfaces if a corner is assigned to more than 3 surfaces, or a edge is assigned to more than 2 surfaces, or a face is assigned to more than 1 surface. GridPro will make a proper reduction of such over assignments. To generate a good grid, it is required that the surfaces of every possible reduction produce the same intersection. For example, if an edge is assigned to 3 surfaces, the 3 intersection curves generated from 3 possible selections of 2 surfaces out of 3 should be the same. If a corner is assigned to 4 surfaces, the 4 surfaces should intersect at one point. One should note that these requirements are not generally satisfied since in general 3 surfaces do not intersect at one point and 4 or more surfaces do not have intersection. Therefore, special care must be taken to insure the requirements are satisfied.

This rule can be overruled by using exclude(`x`), and add(`a`) -statements to eliminate manually the over assignments for the relevant corners, edges and faces.

2.6.2 Rules of valid topology

The rules of valid topology are the rules that the user must follow to design a valid topology. In order to follow these rules, one must first understand the automatic rules used in GridPro (see previous subsection).

A topology that GridPro can successfully parse and compile is a valid topology. GridPro accepts very general topology input. However, GridPro may reject certain seemingly valid topologies for various reasons.

1) *The rule of irreducibility:* With a valid topology, the grid region must be decomposed into full face matching blocks without dangling, unused, or redundant corners, edges or faces.

2) *The rule of connectedness:* All parts of a topology must be connected. That is, a valid topology is one that can not be divided into disconnected parts. If a topology can be divided into disconnected parts, it usually means that the region to be gridded is composed of disconnected subregions. In this case, one should grid each of the subregions independently.

3) *The rule of singularity:* GridPro will reject any topology that has an edge that has more than 8 blocks to it. This rule stems from the grid quality considerations.

4) *The rule of surface closure:* In the topology, the faces assigned to external surfaces must form a full closure of the grid region. That is, these faces should fully separate the region to be gridded from the rest of the world. If internal surfaces are involved, the faces on non-float surfaces is better to cut fully the grid region into disjoint subregions.

These rules are the minimum requirements for a valid topology. However, a valid topology does not always guarantee a grid, let alone a good grid, since the rules here concern only the topological acceptability of the blocking. Many other factors affect whether a grid will be generated. A class of these factors concerns the physical space properties of the involved surfaces and initial corner positions. In the following, we give a few typical situations where a valid topology may lead to bad grids or no grids at all.

1) *Incompatible topology*: The block topology is not suited for the physical shape of the region to be gridded.

2) *Bad initial corner positions*: The initial positions of corners can be generous, but should not be too wild.

3) *Incompatible surfaces*: The physical space properties of surfaces must be compatible with topological requirements. For example, if the topology requires two surfaces intersect, they must intersect truly in the real space. A true intersection of two surfaces is a curve and at every point on the curve, the two normals of the two surfaces are distinct.

4) *Bad internal surfaces*: A bad internal surface can mean either that the surface has too much non-uniformity of curvature, or that the neighbourhoods on the two sides of the surface are too different.

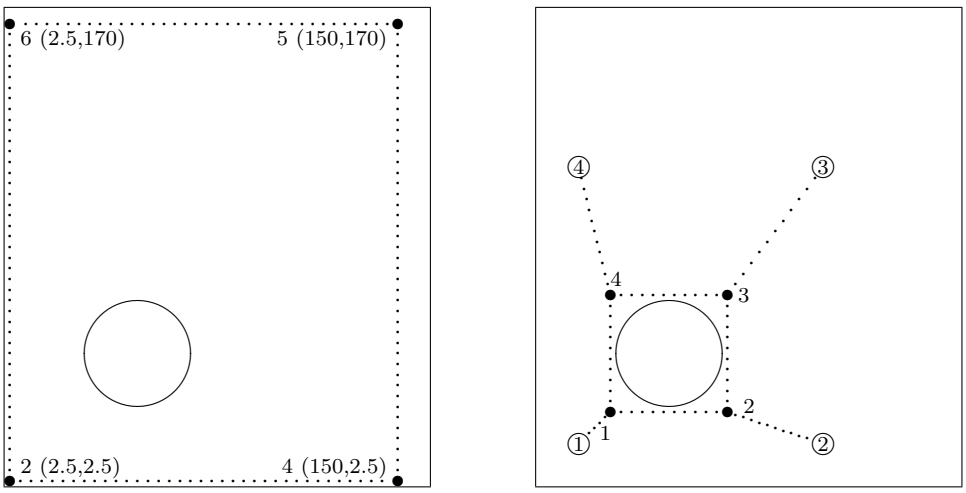
Chapter 3

Using Multiple Levels of COMPONENTs

TIL programs written with multiple levels of *COMPONENT*s are more compact, easier to maintain and modify, and clearer structure-wise. *COMPONENT*s are reusable. Thus *COMPONENT* libraries can be built for general use. Using a *COMPONENT* in TIL is very similar to using a subroutine in FORTRAN. Users should take advantage of this capability of TIL.

3.1 A new look of the old topology

For the same geometry and topology shown in Figure 1.2, we now redo it in a multiple *COMPONENT* style in Figure 3.1. Same as before, the solid lines are surfaces, dotted lines and the big dots are our topology design. We have constructed two *COMPONENT*s in Figure 3.1(a) and (b), and named them *box* and *circle* respectively. The new symbol, a circle with a number in



(a) (b)

Figure 3.1: (a) *COMPONENT box.* (b) *COMPONENT circle.*

it, represents an imported corner from the outside of that *COMPONENT*. The initial positions for corners in the *COMPONENT* circle are not given since we have decided to assign them dynamically, i.e. they will depend on the initial positions of corners imported.

The other thing you might have noticed is that we have assigned the labels 2, 4, 5 and 6 for the corners in *box*. This is just to show you that the labels do not have to start from 1 and there can be gaps in between. You may find this useful when you try to modify an existing topology. For example, if you have to strike out a few corners and make a few new links on an old topology, you can still use those old labels. The TIL program now looks like this,

Program 3.1 *File 'example11.fra', a variation of 'example1.fra'*

```
SET DIMENSION 2
SET GRIDDEN   16

COMPONENT circleInBox()
BEGIN
  s 1   -plane   ( 1.0    0  0     0) ;
  s 2   -plane   (   0  1.0  0     0) ;
  s 3   -plane   (-1.0    0  0 160.0) ;
  s 4   -plane   (   0 -1.0  0 180.0) ;
  s 5   -ellip   (0.05 0.05 0) -t 50.0 50 0 ;

  INPUT 1 box(   sIN  (  1..4), cOUT (2 4..6)) ;
  INPUT 2 circle(cIN  (1:1..4), sIN  (5) ) ;
END

COMPONENT box( sIN  s[0..3] )
BEGIN
  c 2   2.5 2.5 0   -s s:0 s:1 ;
  c 4   150 2.5 0   -s s:1 s:2   -L 2 ;
  c 5   150 170 0   -s s:2 s:3   -L 4 ;
  c 6   2.5 170 0   -s s:3 s:0   -L 5 2 ;
END

COMPONENT circle( sIN  sc, cIN  cb[1..4] )
BEGIN
  c 1   @ 0.99*<cb:1>+0.01*<cb:3>  -s sc  -L cb:1 ;
  c 2   @ 0.99*<cb:2>+0.01*<cb:4>  -s sc  -L cb:2 1 ;
  c 3   @ 0.99*<cb:3>+0.01*<cb:1>  -s sc  -L cb:3 2 ;
  c 4   @ 0.99*<cb:4>+0.01*<cb:2>  -s sc  -L cb:4 3 1 ;
  g   1 cb:1 32;
END
```

Program 3.1 and Program 1.1 represent the same block topology, except for the difference in the corner's initial positions and corner labels (as well as the internal edge and block labels). Thus, Program 3.1 will generate the same final grid as Program 1.1 does, provided they run with the same parameter settings in their schedules. However, the two programs look very different.

3.2 Inputting *COMPONENT*s

The main difference is that the eight corners of *circleInBox* in Program 1.1 are split into two components, *box* and *circle*. The four corners (corners 2, 4, 5 and 6) of *box* in Program 3.1 correspond to corners 5 to 8 in Program 1.1 and the four corners of *circle* in Program 3.1 correspond to corners 1 to 4 in Program 1.1. *CircleInBox* in Program 3.1 inputs component, *box* and *circle* with the two statements,

```
INPUT 1 box( sIN ( 1..4 ), cOUT ( 2 4..6 ) ) ;
INPUT 2 circle( cIN  ( 1:1..4 ), sIN  ( 5 ) ) ;
```

Here, the number following the key word, *INPUT*, is the input label that is assigned for the inputted component. Then, the component name with required argument assignments follows. Same as for the labels of corners and surfaces, the labels for the *INPUT*s in a *COMPONENT* should be assigned with an increasing order starting from a non-negative number. Gaps in the numbering are allowed.

When an *INPUT* statement is processed by GridPro, a copy of the original component is made, manipulated, and placed in the overall topology with proper connections. By original, we mean the part of the TIL program that defines the component. The original of a component is also called the definition of the component. For example, the original copy (or the definition) of *box* is,

```
COMPONENT box( sIN  s[0..3] )
BEGIN
  c 2   2.5 2.5 0   -s s:0 s:1;
  c 4   150 2.5 0   -s s:1 s:2  -L 2;
  c 5   150 170 0   -s s:2 s:3  -L 4;
  c 6   2.5 170 0   -s s:3 s:0  -L 5 2;
END
```

Connections between *box* and *circle* are made by passing relevant surfaces and corners through interfacial array variables.

3.3 Passing-in and out variables

In Program 3.1, the head component, *circleInBox*, after defining five surfaces, first inputs a copy of *box*, with the statement,

```
INPUT 1 box( sIN ( 1..4 ), cOUT ( 2 4..6 ) );
```

It should be compared with the interface defined in the original copy of *box*,

```
COMPONENT box( sIN s[0..3] )
```

The type key, *sIN*, indicates the following variable arguments introduced or evaluated are for passing-in surfaces. Similarly, *cOUT* are for passing-out corners. Two more types, *cIN* and *sOUT*, not used here, are for passing-in corners and passing-out surfaces respectively.

s[0..3] in the *COMPONENT* line above defines an interfacial array variable named *s* with an element range from 0 to 3 for passing-in surfaces. The elements of this array variable can be referenced inside the component *box* in the way explained in the next section. A variable name

should have no more than 8 characters starting with a letter followed by any combination of letters, digits, and the character '_'. Upper-case and lower-case letters are distinct.

As a rule of TIL, all passing-in variables must be explicitly defined in the component interface of the original copy as variable arrays, each with a name and range.

On the other hand, no passing-out variable should be explicitly defined in the component interface of original copy. They are defined and assigned with values in the component interface of an *INPUT* statement.

In Program 3.1, when a *box* is inputted in another component *circleInBox*, the array variable *s[0..3]* must be assigned values with a matching type and length in the component interface of the *INPUT* statement.

A pair of parentheses indicates the assignment (or evaluation) of a variable array. For our case, the assignment *sIN (1..4)* in the *INPUT* statement matches in length (4 elements) and type (*sIN*) with the variable *s[0..3]* defined in the interface of the original *box*. The elements 0 to 3 of variable *s* are assigned the values of surfaces 1 to 4 of *circleInBox*. That is, inside the current copy of *box*, a reference to the 0^{th} element of variable *s* means surface 1 in *circleInBox*, and a reference to the 1^{st} element of variable *s* means surface 2 of *circleInBox*, and so forth.

Notice that the variable name *s* does not appear in the assignment. In general, variable names do not appear with the assignments for passing-in variables since the position of the pair of assignment parentheses from the type key *sIN* or *cIN* uniquely determines which variable is being assigned. In this way, the first pair of parentheses following *sIN* in an *INPUT* statement indicates the assignment of the first variable array following *sIN* in the corresponding original *COMPONENT*, the second pair of parentheses following *sIN* indicates the assignment of the second variable array following *sIN* and so on so forth until a new type key or no more pairs of parentheses are found.

Similarly, corners 2,4,5 and 6 of *box* are passed out from *box* with the variable assignment *cOUT (2 4..6)* in the *INPUT* statement above. Here *cOUT (2 4..6)* introduces to the parental component, *circleInBox*, a new variable of type *cOUT* and length four with a range from 1 to 4 and a default name (Note: the ranges of variables introduced with *cOUT* or *sOUT* always start from 1). The elements 1 to 4 of this variable are assigned the values of corners 2,4,5 and 6 of *box*. Thus, the first element of the variable (which is referenced with 1:1) means corner 2 in *box* after the assignment, and the second element of the variable (which is referenced with 1:2) means corner 4 of *box*, *etc.* For more detail about variable referencing, see the next section.

A name can be associated with a passing-out variable. For example, if we have *cOUT corn4(2 4..6)* instead of *cOUT (2 4..6)*, the variable automatically introduced will have a name *corn4* instead of a default name. For an *INPUT* statement, there can only be at most one variable with a default name; usually it is the first variable following the type key *cOUT* or *sOUT*. Names for non-default named variables in an *INPUT* statement must be distinct. However, the same name can be used for variables in different input statements. When referenced, they will be distinguished by the input labels of the variable referencing mechanism.

Since an assignment for a passing-out variable automatically defines a new variable, different copies of the same component can have different passing-out variables.

3.4 Referencing variables

A major difference between Program 3.1 and 1.1 is the concept of a variable being used as was done in Program 3.1 and not in Program 1.1. We have already touched on some aspects of it in the last section. We will give a more complete discussion here.

First, all variables in TIL are array variables, of which the values are numbers used to label the corners or surfaces. To be simple, one can regard the labels of the corners or surfaces as pointers to the corners or surfaces.

An array variable is always associated with an element range which can be as short as 1. The same corner in two different copies of the same component should be regarded as two distinct corners at any of their parental component levels. Therefore, they will be referenced differently.

Second, variables can be introduced or defined explicitly or implicitly. All the passing-in variables for a component are explicitly defined in the interface of the component definition (also called the original copy). All the passing-out variables for a component are implicitly introduced when a copy of the component is inputted. Thus, one can introduce different passing-out variables for the same component at different *INPUT*s.

Two more variables are implicitly defined. One is for local corners. Local corners are those defined with a corner definition statement in the current component. The other is for local surfaces. A TIL program line like,

```
c 6   150 2.5 0  -s 2 3  -L 2 5;
```

is actually assigning the 6^{th} element of the local corner variable array.

In fact, TIL considers any reference to a corner or surface as a variable referencing. Thus, the numbers, 2 and 3, following *-s* in the above TIL program line actually refer to the 2^{nd} and 3^{rd} elements of the local default surface variable; The numbers, 2 and 5, following *-L* refer to the 2^{nd} and 5^{th} elements of the local default corner variable. The most general form of variable referencing is,

input_label var_name:num1...num2

As we have seen, not all parts are needed for a variable referencing. The following is a complete list of examples for variable referencings:

3corn4:5..10	– the 5^{th} to 10^{th} elements of the variable *corn4* passing-out from the inputted component 3.
3:5..10	– the 5^{th} to 10^{th} elements of the default variable passing-out from the inputted component 3.
corn4:5..10	– the 5^{th} to 10^{th} elements of the variable *corn4* defined in the interface of the current component.
5..10	– the 5^{th} to 10^{th} elements of the local default variable.
3corn4:5	– the 5^{th} element of the variable *corn4* passing-out from the inputted component 3.
3:5	– the 5^{th} element of the default variable passing-out from the inputted component 3.
corn4:5	– the 5^{th} element of the variable *corn4* defined in the interface of the current component.
5	– the 5^{th} element of the local default variable.
corn4	– the 1^{st} element of the variable *corn4* defined in the interface of the current component.

Using these examples, you should be able to interpret easily the meaning of variable referencings used in Program 3.1.

3.5 Relatively positioning corners

The last important difference between Program 3.1 and 1.1 is the use of relative positioning of corners in *circle*. This is done by evaluating a vector expression at the time when the component is inputted. The initial positions of all the corners in *circle* are setup in this manner. To see it more clearly, consider the following statement in *circle*,

c 1 @ 0.99*<cb:1>+0.01*<cb:3> -s sc -L cb:1 ;

In the place for an initial position, we find,

@ 0.99*<cb:1>+0.01*<cb:3>

The flag @ indicates that the initial position of this corner is specified as the value of the vector expression next to it evaluated at the input time. The vector expression, *0.99*<cb:1>+ 0.01*<cb:3>*, results in a vector which is '*0.99 times the position vector of the corner referenced by the* 1^{st} *element of array variable* cb *plus 0.01 times the position vector of the corner referenced by the* 3^{rd} *element of array variable* cb'. Here, < *corner_reference* > means the initial position vector of the *'corner_reference'*d. A vector can also be given in the form with the coordinates specified, such as {*1.1,2,0*}.

A complete discussion of vector expressions is left to Chapter 5. A rule that concerns the evaluation of an expression is: when the current component is a transformed input (translation, rotation, scaling,...) at the parental level, only directly positioned (without vector expressions involved) corners, surfaces and inputs in the current component will be transformed.

3.6 Moving *COMPONENT*s around

There are many ways to program the same topology. This adds the flexibility to best suit the user's needs. To prepare for the needs of the example in the next section, we reprogram the topology design in Figure 3.1 as follows:

Program 3.2 *File 'example12.fra', another variation of 'example1.fra'.*

```
SET DIMENSION 2
SET GRIDDEN   16

COMPONENT circleInBox()
BEGIN
  s 1   -plane ( 1.0    0  0     0) ;
  s 2   -plane (   0  1.0  0     0) ;
  s 3   -plane (-1.0    0  0 160.0) ;
  s 4   -plane (   0 -1.0  0 180.0) ;

  INPUT 1 box(    sIN  (  1..4), cOUT (2 4..6)) ;
  INPUT 2 (50 50 0)*circle( cIN  (1:1..4)) ;
END

COMPONENT box( sIN  s[0..3] )
BEGIN
  c 2   2.5 2.5 0   -s s:0 s:1 ;
```

```
  c 4   150 2.5 0   -s s:1 s:2   -L 2 ;
  c 5   150 170 0   -s s:2 s:3   -L 4 ;
  c 6   2.5 170 0   -s s:3 s:0   -L 5 2 ;
END

COMPONENT circle( cIN  cb[1..4] )
BEGIN
  s 1   -ellip (0.05 0.05 0) ;

  c 1   -20 -20 0   -s 1   -L cb:1 ;
  c 2    20 -20 0   -s 1   -L cb:2 1 ;
  c 3    20  20 0   -s 1   -L cb:3 2 ;
  c 4   -20  20 0   -s 1   -L cb:4 3 1 ;
  g   1 cb:1  32;
END
```

In comparison with Program 3.1, the most significant change is in *circle*. The main thing we want to illustrate is how you can input a component with transformed positions from the original copy and the effects of such transformations.

Consider the input statement in Program 3.2,

```
INPUT 2 (50 50 0)*circle( cIN  ( 1:1..4 ) ) ;
```

It says '*input a copy of* circle *and put it at a position which is translated by the vector (50,50,0) from the position of the original copy*'. Here (50 50 0) represents a translation operator and the character '`*`' means '*operate on*'. The precise effect of this operation is: everything involving real space positions inside *circle* except those associated with passing-in variables will be translated by the vector (50, 50, 0). Things that involve real space positions are either initial corner positions or surface locations.

Another operator you can use is a linear transformation (rotation, stretching or reflection). Consider an input statement,

```
INPUT 2 (0.5 0.861 0 0.861 -0.5 0 0 0 1)*
circle( cIN  ( 1:1..4 ) ) ;
```

In this example, a position vector $\begin{pmatrix} x \\ y \\ z \end{pmatrix}$ in *circle* will be transformed to,

$$\begin{pmatrix} 0.5x + 0.861y \\ 0.861x - 0.5y \\ z \end{pmatrix} = \begin{pmatrix} 0.5 & 0.861 & 0 \\ 0.861 & -0.5 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

You can also use more than one operator as shown in the following example,

```
INPUT 2 (0.5 0.861 0 0.861 -0.5 0 0 0 1)*(50 50 0)*
circle( cIN  ( 1:1..4 ) );
```

When you use multiple operators, a right one will operate on the component before those on its left. Thus, in the above example, *circle* is first translated and then rotated.

Notice, the surface definition for the circle becomes a local definition in *circle* instead of passing-in from the interface of *circle*. Notice also, the *-t...* option is, now, gone from this

surface. The reason is that this translation is done when *circle* is inputted in *circleInBox*. A consequence of the locally defined circle surface is that every time *circle* is inputted, a copy of the circle surface is defined for it. This can be very useful. For example, you can build a *COMPONENT* called *jetEngine* with the engine geometry locally defined in it, and assemble multiple copies of this *jetEngine* into the topology design for an airplane without multiply defining the engine geometry.

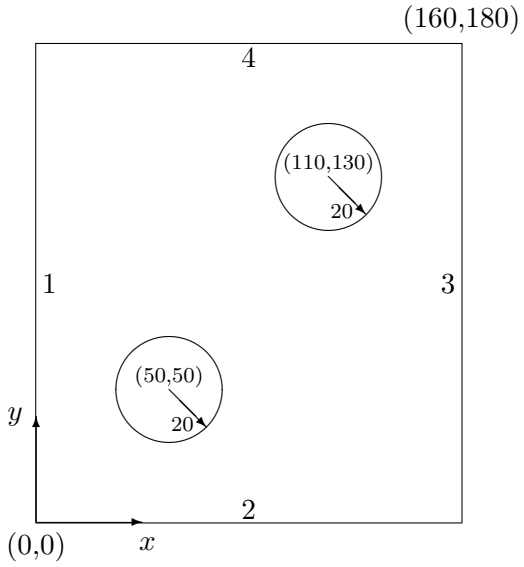


Figure 3.2: Geometry of the region to be gridded.

A more general form of transformation is specified by using vector expressions. In this case, only one operator can be applied to an *INPUT*. For more detail, see Chapter 5.

3.7 Including and Reusing old *COMPONENT*s

Ok, after so many words, you may ask: Is it worth so much trouble to grid such a simple case which is basically a problem of single block? Well, let us do more with what we have already gotten. Along with it, we will show you how you can include and reuse components.

Consider adding one more circle of the same size into the configuration in Figure 1.2(a). Figure 3.6 shows the new configuration. The region to be gridded is outside the circles and inside the box. The two circles are not assigned with surface labels since the surfaces are defined inside *circle* of Program 3.2 which will be used in our new program without changes.

Instead of *box*, we will design a bit more complicated component called *boxes* using a lower level component *line*. The two components are shown in Figure 3.7. *COMPONENT boxes* is built by making six copies of *line* with proper vertical translations and connections. The corners in Figure 3.7(b) are not labeled with numbers since they are all from the lower level component *line*. The variable referencing labels in Figure 3.7(b) are for passing-out corner variables. Later, they will be used to connect two copies of *circle* to *boxes*.

Now, let us say that *line* and *circle* are two basic components that will be used over and over again (in reality this may not be true!). So we put them together in a file called '*lib2d.fra*'. Program 3.3 lists '*lib2d.fra*'.

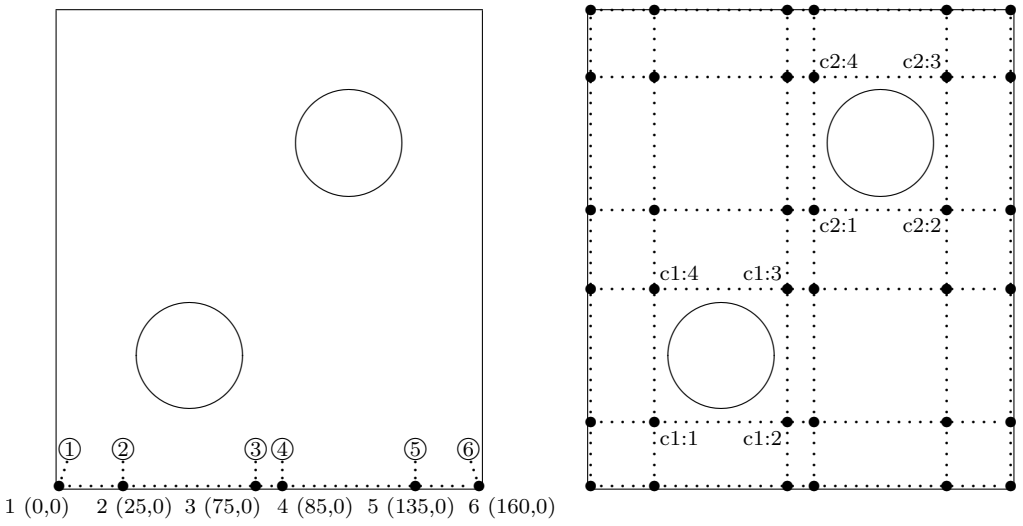


Figure 3.3: (a) *COMPONENT line*. (b) *COMPONENT boxes* built from *line*.

Program 3.3 *A component library file* 'lib2d.fra'

```
COMPONENT line(sIN sx[1..2], sy, cIN pl[1..6])
BEGIN
  c 1      0 0 0  -s sy sx:1 -L pl:1 ;
  c 2     25 0 0  -s sy      -L pl:2 1 ;
  c 3     75 0 0  -s sy      -L pl:3 2 ;
  c 4     85 0 0  -s sy      -L pl:4 3 ;
  c 5    135 0 0  -s sy      -L pl:5 4 ;
  c 6    160 0 0  -s sy sx:2 -L pl:6 5 ;
END

COMPONENT circle( cIN  cb[1..4] )
BEGIN
  s 1  -ellip (0.05 0.05 0) ;
  c 1   -20 -20 0   -s 1  -L cb:1 ;
  c 2    20 -20 0   -s 1  -L cb:2 1 ;
  c 3    20  20 0   -s 1  -L cb:3 2 ;
  c 4   -20  20 0   -s 1  -L cb:4 3 1 ;
  g  1 cb:1 32;
  x f     1 3     cb:1 cb:3 ;
END
```

Here *circle* is the same as in Program 3.2. The TIL program for the component *boxes* is listed below,

Program 3.4 *A part of file 'example2.fra'*

```
COMPONENT boxes()
BEGIN
  s 1 -plane  ( 1.0    0  0     0) ; #x1 side
  s 2 -plane  (   0  1.0  0     0) ; #y1 side
  s 3 -plane  (-1.0    0  0 160.0) ; #x2 side
  s 4 -plane  (   0 -1.0  0 180.0) ; #y2 side
  INPUT 1
   line(sIN (1 3),( 2),cIN (-6)    ,cOUT (1..6));
  INPUT 2 (0  25 0)*
   line(sIN (1 3),(-1),cIN (1:1..6),cOUT (1..6));
  INPUT 3 (0  75 0)*
   line(sIN (1 3),(-1),cIN (2:1..6),cOUT (1..6));
  INPUT 4 (0 105 0)*
   line(sIN (1 3),(-1),cIN (3:1..6),cOUT (1..6));
  INPUT 5 (0 155 0)*
   line(sIN (1 3),(-1),cIN (4:1..6),cOUT (1..6));
  INPUT 6 (0 180 0)*
   line(sIN (1 3),( 4),cIN (5:1..6),cOUT (1..6));
END
```

Here a negative number in a pair of variable assignment parentheses means assigning NULLs to certain elements of the variable. For example, (-6) in the line, '*INPUT 1 line(...)*' means assigning a NULL to each of the first six elements of variable *pl* of *line*. A mixed assignment of NULLs and non-NULLs is allowed. However, an assignment must match in length with the variable.

To include the topology design for two circles, the file 'example2.fra' now looks like this,

Program 3.5 *File 'example2.fra'*

```
SET DIMENSION 2
SET GRIDDEN   6
INCLUDE   "lib2d.fra"

COMPONENT circ2InBox()
BEGIN
  INPUT 1 boxes(cOUT c1(2:2 2:3 3:3 3:2),
                    c2(4:4 4:5 5:5 5:4));
  INPUT 2 ( 50  50 0)*circle(cIN (1c1:1..4));
  INPUT 3 (110 130 0)*circle(cIN (1c2:1..4));
END
```

[put Program 3.4 here]

Since the components *line* and *circle* are in the file, "lib2d.fra", it is included with the line, '*INCLUDE "lib2d.fra"*'.

The include-line, '*INCLUDE "lib2d.fra"*' constitutes the include section of the TIL program. In general, the include section can contain more than one include-line, each with a different file.

When an included file is processed, all global assignments are ignored; Component definitions are taken in; And include lines in the included file are further processed.

To generate a grid, you need to have your run schedule written in the file '*example2.sch*'. For a simple run, this file can be an exact copy of '*example1.sch*'. The grid generated from 'example2.fra' with a schedule the same as 'example1.sch' is shown in Figure 3.4.

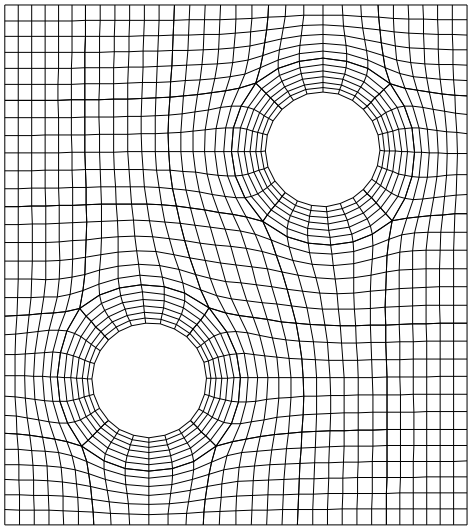


Figure 3.4: A grid generated from 'example2.fra'

Chapter 4

Modifying An Existing Topology

To generate a grid satisfying certain requirements, one may design and try out different topologies. Quite often, a newer topology differs with an older one by some slight local change. TIL provides the means to edit your topology. This includes deleting, adding and modifying operations for the defined surfaces and corners, and for the generated edges, faces and blocks.

The need of topology modification arises also from the need to specify topologies by modifying those generated from the default rules in GridPro. For example, an `'x f'` statement can be used to delete some faces that are otherwise automatically generated by GridPro.

The statements for the modification of existing definitions of corners, edges, faces and blocks have the general structure that starts with an action key (`x` – for excluding or `a` – for adding) and an object key (`e` – for edges, `E` – for surfaces of edges,...) followed by a list of corner and surface referencing.

The corner and surface references used in such statements must, as usual, be defined before the statement. However, for a variable reference, a value NULL is allowed. A non-existing object (edge, face, or block) defined by corners in the list is not an error and the corresponding corners and surfaces will be ignored.

In what follows, the notations such as Block(4, 6), Face(4, 6), and Edge(4, 6) will be used. The notation Block(4, 6) is used to mean the block that has corners 4 and 6 as a pair of diagonal corners. Face(4, 6), and Edge(4, 6) are similarly defined.

4.1 Deleting things from your topology

The `x`-statement is designed to delete specific topological elements that are generated by the default rules of GridPro. These pertain to unwanted corners, edges, faces, blocks, and boundary conditions. We will use examples to illustrate the use of the `x`-statement.

Deleting corners

Example:

```
x c 3 c:2 5;
```

This statement deletes corners 3, c:2 and 5. When a corner is deleted, all the links to it are also deleted.

Deleting edges

Example:

```
x e 3 c:2 5 4;
```

This statement deletes Edge(3, c:2) and Edge(5, 4). The corners listed must be in pairs.

Deleting faces

Example:

```
x f 3 c:2 5 4;
```

This statement deletes Face(3, c:2) and Face(5, 4). The corners listed must be in pairs.

Deleting blocks

Example:

```
x b 3 c:2 5 4;
```

This statement deletes Block(3, c:2) and Block(5, 4). The corners listed must be in pairs.

Deleting surfaces from corners

Example:

```
x C 3 c:2 5 4;
```

This statement deletes surface 4 from corners c:2, 3 and 5

Deleting surfaces from edges

Example:

```
x E 3 c:2  5 4 s:2;
```

This statement deletes surface s:2 from Edge(3, c:2) and Edge(5, 4).

Deleting surfaces from faces

Example:

```
x F 3 c:2   5 4 s:2;
```

This statement deletes surface s:2 from Face(3, c:2) and Face(5, 4).

4.2 Adding things to your topology

The `a`-statement is used to add different items to corners, edges, faces, and blocks generated by GridPro with default rules. We will again use examples to illustrate the use of the `a`-statement.

Adding edges

Example:

```
a e 3 c:2 5 4;
```

This statement adds a link from corner 3 to corner c:2, and a link from corner 5 to corner 4. If the link exists already, no additional link is added. The corners listed must be in pairs.

Adding surfaces to corners

Example:

```
a C 3 c:2 5 4;
```

This statement adds surface 4 to corners 3, c:2 and 5. If the surface exists already for the corner, no additional surface is added.

Adding surfaces to edges

Example:

```
a E 3 c:2  5 4  s:2;
```

This statement adds surface s:2 to Edge(3, c:2) and Edge(5, 4). If the surface exists already for the edge, no additional surface is added.

Adding surfaces to faces

Example:

```
a F 3 c:2  5 4  s:2;
```

This statement adds surface s:2 to Face(3, c:2) and Face(5, 4). If the surface exists already for the face, no additional surface is added.

4.3 Changing a corner's position

A corner position can be changed by a `<..>` -statement of TIL. Consider the example:

```
<c:3> = 0.5*(<c:1> + <c1:2>) ;
```

The structure here is

< *corner_ref* > = *vector_expr* ;

This example statement sets the initial position of corner c:3 to the value calculated from `0.5*(<c:1> + <c1:2>)` which is the middle point of the initial positions of corners c:1 and c1:2.

The reason of using `<..>` -statements here is that for certain cases whether a run can be successful depends on the initial positions of corners in certain sensitive areas where grid foldings can happen easily.

Chapter 5

Using Vector Expressions to Define Positions

Vectors and vector expressions present a useful means to setup initial corner positions and to put geometries in place.

5.1 Declaring vectors

Vectors can be declared inside a `COMPONENT`, and vector declarations must appear before any statement in the `COMPONENT`. A vector declaration starts with the type keyword `VECTOR` followed by one or more vector variable names with or without range specifications. A `COMPONENT` may have more than one vector declaration. Let's look at an example vector declaration section of a `COMPONENT`,

```
COMPONENT junk()
BEGIN
   VECTOR x[3..10],r;
   VECTOR ix;
.
.
.
```

The first vector declaration declares a vector array named `x` with its element range from 3 to 10, and a vector variable named `r`. The second vector declaration declares a single vector variable `ix`.

A vector behaves very much like a corner as far as the position attribute of a corner is concerned. Therefore, vectors can be passed in and out of `COMPONENT`s in the same fashion as corners do; Likewise, corners can be used in and evaluated with vector expressions, just as vectors do. Hence, unless explicitly stated, the phrase "a vector variable" in this manual can mean either the position attribute of a corner variable or a vector variable defined with a `VECTOR` declaration.

Since a variable in a `COMPONENT` interface is passed by reference (FORTRAN - like) rather than by its value, a change to a vector's value in a child level will be carried back to the parent level upon the return from the child `COMPONENT`.

5.2 Where can vector expressions be used ?

Vector expressions can be used to define a physical space position in various places in a TIL program. For all these situations, except evaluating a vector with a "`< > =...`" statement, a character '@' indicates the use of vector expressions. One needs to remember that both using vector expressions in a `COMPONENT` and using transformation operators when `INPUT`ting this `COMPONENT` the same time without understanding the interaction between them may cause unexpected results. In the following subsections, we sequentially examine the uses of vector expressions. The above mentioned interaction will be also discussed.

5.2.1 Evaluating a vector variable

A vector variable can be evaluated by a `<`*var*`> =`*vexpr* statement.

`Example:`

```
<c:1> =  {1.2,0,0} + <x> + <y>;
```

It evaluates the vector variable `c:1` to be the sum of the three vectors in the vector expression on the right hand side of the equal sign, namely `{1.2,0,0}`, `<x>` and `<y>`.

If `c:1` is, in fact, a corner variable, this statement changes the initial position of the referenced corner.

5.2.2 Setting a corner's position

A corner's initial position can be set by using a vector expression in the place of the initial coordinates in the `c`-statement.

`Example:`

```
c 3 @ <ctr> + <x> - <y>  -s 1  -L 2;
```

The symbol '@' here indicates that a vector expression follows. The initial position of corner 3 is set to the resulting value of evaluating the vector expression.

To stress the difference between the two methods of defining the initial position of a corner, let's examine the following segment of a TIL program,

```
c 3    1 2 3   -s 1  -L 2;
c 4  @{1,2,3}  -s 1  -L 3;
```

On the first look, one may think that corners 3 and 4 have the same initial position. However, this observation may or may not be correct, and it all depends on whether the `COMPONENT` containing the two statements is transformed or not when it is `INPUT`ted. The initial position of corner 3 is transformed by the same transformation operations that are acting on the `COMPONENT`; while the initial position of corner 4 is not affected by them. In fact, this is a general rule for all vector expressions.

5.2.3 Define some built-in implicit surfaces

Two of the built-in implicit surfaces can be defined with vector expressions. For a surface of type `-plane`, one can, instead of using `-plane(`*a b c d*`)`, use

`-plane@(`*norm_vexpr,point_vexpr*`)`

Here *norm_vexpr* is the normal vector of the plane and *point_vexpr* is a point on the plane.

`Example:`

```
s 2 -plane@(<x>^<y>, {1,0,0});
```

This statement defines a plane surface that has the cross product `<x>^<y>` as its normal vector and it passes through the point `{1,0,0}`.

The parameters for a surface of type `-ellip` can be set with

`-ellip@(`*u_vexpr, v_vexpr, w_vexpr, t_vexpr*`)`.

where *u_vexpr, v_vexpr* and *w_vexpr* are the three semi-axes of the ellipsoid and *t_vexpr* is the center of the ellipsoid. To be more precise, let's denote **U**, **V**, **W** and **T** to be the resulting vectors of *u_vexpr, v_vexpr*, *w_vexpr* and *t_vexpr* respectively and denote $\mathbf{U}_x$, $\mathbf{U}_y$ and $\mathbf{U}_z$ to be the three components of vector **U**, and so forth.

The surface is now defined by the equation,

$$u^2 + v^2 + w^2 = 1.$$

where (u, v, w) is obtained via solving,

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \mathbf{U}_x & \mathbf{V}_x & \mathbf{W}_x \\ \mathbf{U}_y & \mathbf{V}_y & \mathbf{W}_y \\ \mathbf{U}_z & \mathbf{V}_z & \mathbf{W}_z \end{pmatrix} \begin{pmatrix} u \\ v \\ w \end{pmatrix} + \begin{pmatrix} \mathbf{T}_x \\ \mathbf{T}_y \\ \mathbf{T}_z \end{pmatrix}.$$

5.2.4 Transformation operators for surfaces

When vector expressions are used for the transformation of surfaces, only the '`-t`' operator is allowed ('`-R`' is not allowed). In this case '`-t`' means a general transformation, instead of just a translation. The arguments for a '`-t`' operator have a general form as follows:

`-t @(`*t_vexpr*, *x_vexpr*, *y_vexpr*, *z_vexpr*`)`

Here *t_vexpr* is the translation part of the transformation, and *x_vexpr*, *y_vexpr* and *z_vexpr* are the rotation (and stretching) part of the transformation. A short form of the operator can be used:

`-t @(`*t_vexpr*`)`

In this case, the rotation (and stretching) part of the transformation is an identity matrix.

Let's denote **X**, **Y**, **Z** and **T** to be the resulting vectors of *x_vexpr, y_vexpr*, *z_vexpr* and *t_vexpr* respectively and denote $\mathbf{X}_x$, $\mathbf{X}_y$ and $\mathbf{X}_z$ to be the three components of the vector **X**, and so forth. Let (x, y, z) and (x', y', z') be the corresponding surface points before and after the transformation respectively. Then, (x, y, z) and (x', y', z') are related by the equation,

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} \mathbf{X}_x & \mathbf{X}_y & \mathbf{X}_z \\ \mathbf{Y}_x & \mathbf{Y}_y & \mathbf{Y}_z \\ \mathbf{Z}_x & \mathbf{Z}_y & \mathbf{Z}_z \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} \mathbf{T}_x \\ \mathbf{T}_y \\ \mathbf{T}_z \end{pmatrix}.$$

An example is shown below,

```
s 3 -linear "surf1.dat" -t @({1,2,3}, <x1>^<x2>, 2*<y>, <z>) ;
```

5.2.5 Transformation operators for INPUTs

The transformation operators using vector expressions for an INPUT statement have the same format as for an s-statement in the previous subsection, except that only the long form is allowed. The corresponding points before and after the transformation also have the same relation as in the previous subsection. The following line is an example TIL line,

```
INPUT 3 @({1,2,3}, <x1>^<x2>, 2*<y>, <z>) * test(cIN (c:1..8)) ;
```

5.3 Vector and scalar expressions

Scalar expressions are used to construct vector expressions. Some commonly used operators are defined in GridPro. The precedence and associativity of these operators are also as commonly defined. When it is not clear, a good practice is to use parentheses to group things.

The following operators can be used for vector expressions:

`(`*vexpr*`)` – Grouping; Has the highest precedence.
sexpr `*` *vexpr*
vexpr `*` *sexpr* – Scalar product.
vexpr `^` *vexpr* – Cross product.
`[`*vexpr*`]` – Normalizing a vector.
`far(`*vexpr*`)` – get the furthest axis from *vexpr*.
vexpr `+` *vexpr* – Vector sum.
vexpr `-` *vexpr* – Vector subtraction.
`-`*vexpr* – Reversing the vector direction.
`<`*var*`>` – referencing a vector variable or the position of a corner variable.
{*x_sexpr*, *y_sexpr*, *z_sexpr*} –Vector with its components evaluated via scalar expressions.

The following operators can be used for *sexpr*:

(*sexpr*) – Grouping.
sexpr `*` *sexpr* – Product.
vexpr `*` *vexpr* – Dot product of two vectors.
`|`*vexpr*`|` – Length of a vector.
(*vexpr*)`.x` – x component of a vector.
(*vexpr*)`.y` – y component of a vector.
(*vexpr*)`.z` – z component of a vector.
sexpr `/` *sexpr* – Scalar division.
sexpr `+` *sexpr* – Scalar sum.
sexpr `-` *sexpr* – Scalar subtraction.
`-`*sexpr* – unary minus.
d – a real number.
sin(*sexpr*), cos(*sexpr*), sqrt(*sexpr*), min(*sexpr*1,*sexpr*2)
max(*sexpr*1,*sexpr*2) – as usual.

To illustrate the use of vector and scalar expressions, let's look at some examples:

`<x>` – the vector value of the vector variable `x`.
`{1, 2, 3}` – vector by specifying the components.
`<c:1>^<x>` – cross product of two vectors `c:1` and `x`.
`{(<x>).z, 1.2,cos((<y>).x)}`
– a vector with its x-component=z-component of `x`, y-component=1.2, and z-component = cos(x-component of `y`).
`0.8*<c:1> + 0.2*{1,1,2}`
– 0.8 times vector `c:1` plus 0.2 times vector {`1,1,2`}.

5.4 Mixed use of vector expressions and INPUT transformations

Mixing the use of vector expressions and `INPUT` transformations provides the good things from both worlds, namely the flexibility of vector expressions and the simplicity of `INPUT` transformations. However, pitfalls do exist, unless one understands how GridPro treats this kind of mixing, since what GridPro will do in this situation may or may not be the same as what one might think GridPro should do, though GridPro has adopted the rules as intuitively as possible.

A TIL code with multiple levels of `INPUT`s has a simple tree structure. In the process that GridPro parses a TIL code, it is expanded into a linear chain of topology construction operations, such as corner creation and surface loading. After a corner or vector come into existence, it may have different current coordinate positions at different points on this linear chain.

The corners and vectors are handled differently:

1) A corner defined in a `COMPONENT` gets a new copy and id every time the `COMPONENT` is `INPUT`ted; while a vector defined in a `COMPONENT` shares the same memory for all the `INPUT`s of the `COMPONENT`. For example, look at the following segment of TIL code,

```
COMPONENT junk()
BEGIN
 INPUT 1 test(cOUT corn(c),vect(v));
 <1corn> = {1,0,0};
 <1vect> = {2,0,0};
 INPUT 2 test(cOUT corn(c),vect(v));
 PRINT(‘‘1corn=(%v),2corn=(%v),1vect=(%v),2vect=(%v)\n’’,
         1corn,2corn,1vect,2vect);
.
.
END

COMPONENT test()
BEGIN
 VECTOR v;
 <v> = {0,0,1};
 c 1 @{0,0,0};
END
```

When it is run, one would get the screen print out,

```
1corn=(1 0 0),2corn=(0 0 0),1vect=(0 0 1),2vect=(0 0 1)
```

2) When GridPro is finished with processing an `INPUT`, the transformation on the `INPUT` is applied to the current coordinate position of all the corners created under the given `INPUT` to define the new current coordinate position; while all the vectors are left un-transformed.

3) All the vector expressions are evaluated using the current coordinate positions of its constituents.

A safe practice is that whenever an `INPUT` transformation is used do not pass in any vectors and do not use passing-in corners for vector expressions in the `INPUT`ted `COMPONENT`(you may still use passing-in corners for linking purposes).

Chapter 6

Doing I/O In TIL Programs

To start, let's first make a distinction between different types of I/Os. The I/O operations of TIL programs are not the same as the I/O operations of GridPro as a whole unit. While the later specifically means the input and output of grid data and connectivity data, the former is for inputting and outputting parameters to set up and debug the topology.

This chapter is concerned with the former type of I/O operations. These operations provide more flexibility and reusability to the topology designed. For example, a topology design can be used for configurations with different scales simply by reading in a few scale parameters without the need to change things inside the TIL code. Five types of statements with key words `OPEN`, `CLOSE`, `READ`, `WRITE` and `PRINT` respectively are used for I/O operations. They can appear in different levels of components.

6.1 `OPEN` and `CLOSE` files

Before a file can be accessed in a TIL program, it must be opened with an `OPEN` statement, which has the following syntax:

`OPEN(`*unit*`, "`*file_name*`", `*access_mode*`);`

Here, *unit* is an integer labeling the opened file. It is global in the sense that an opened *unit* in one component can be referred in another component and it stays open until it is explicitly closed with a `CLOSE` statement. The *unit* is a number with a value from 1 to 64 and up to 64 files can be opened simultaneously. To reopen an opened unit is an error.

The second argument specifies the name of the file to be opened. It must be in double quotes.

The *access_mode* is one of the three key words, `READ`, `WRITE` and `APPEND`. For the `READ` mode, attempting to open a nonexisting file is an error. If the file does not exist under the `APPEND` mode, the effect is the same as for `WRITE`.

An opened file can be closed with a `CLOSE` statement,

`CLOSE(`*unit*`);`

A closed *unit* can be reused in future `OPEN`s.

6.2 READ and WRITE (and PRINT) files

PRINT is very similar to WRITE except that the writing is on the terminal screen. Both READ and WRITE require a *unit* number as the first argument. If the *unit* is not opened, the I/O will read from the keyboard and/or write to the screen.

Only vectors can be read with a READ statement under the current implementation. Every READ causes the opened file to advance one line.

Let's examine the sample statement,

```
READ(3,"%v %v %v %v",c:1,2,1:1,v);
```

The first argument 3 is the unit number. The string in the double quotes is a conversion format for the current line of unit 3. The rest of the arguments are corner or vector variables.

Each %v in the conversion format converts a group of three numbers in the current line to a vector and assigns it to the corresponding subsequent argument.

A WRITE statement has a similar syntax except that it has more types of conversions. The following statement is an example,

```
WRITE(4,"Hi:pos=(%v),id=%m\n trace= %t \n",c:1,c:2,c:3);
```

With this statement, two lines are written to unit 4. The format string contains conversion symbols %v, %m and %t, and some other characters. In the output produced with this statement, the conversion string %v is replaced by the position vector of the first variable c:1 in the variable list following the format; %m is replaced by the internal id of c:2; and %t is replaced by the trace of c:3. Of course, the internal id and trace information are only meaningful for corners. Each \n is a new-line symbol in the output. Therefore, the above statement produces two lines in unit 4. They may look like this,

```
Hi:pos=(1 3.435 2.2), id=312
trace= 2:1:5:7
```

Chapter 7

Surface Specifications

This Chapter will be mainly concerned with surface specifications and their relation to surface definitions in the TIL programs. Some topics related to surfaces, such as, internal surfaces, overlapping surfaces and surface assignment rules will not be discussed here (see Chapter 2).

7.1 Surface classifications

7.1.1 Surface types

Every surface definition statement in a TIL program contains a flag for surface type. Surface types are used to indicate the format and structure of surface data. There are seven surface types in the current implementation of GridPro. A surface type belongs to one of the two group types used in GridPro.

The first group type is implicit (analytic) type. In this case, a surface is defined as an equal potential surface of a scalar valued analytic function of the position vector (x, y, z). Some of the simple forms of the functions are hard wired into GridPro. They are called built-in implicit types and are listed below,

```
-plane       --- plane surfaces.
-ellip       --- (super) ellipsoid surfaces.
-xpolar      --- used for periodic BC in polar coordinates.
-xyz         --- used for periodic BC in cartesian coordinates.
```

For non-builtin implicit surfaces, there is a type,

```
-implic      --- general implicit  surfaces.
```

The second group type consists of explicit surfaces. This group type can also be called the parametric type for the reason that a surface point can be determined by the values of a set of parameters.

In current implementation, it contains the following surface types,

```
-linear      --- a) single patch bilinear parametric surfaces.
                 b) surfaces composed of multiple patches of bilinear
                    parametric surfaces.
-quad        --- surface of unstructured quadrilateral elements.
-tria        --- surface of unstructured triangular elements.
-tube        --- surface of revolution around a center curve.
```

A surface of this group is usually specified by a fair amount of data stored in a separate file(s).

7.1.2 Boundary modes

Boundary modes distinguish between different constraints on surfaces. Boundary modes are not explicitly specified in TIL programs. Whether a surface is of a certain boundary mode is determined by how the surface is used in GridPro. There are three boundary modes in GridPro: the fixed-surface mode, the periodic-surface mode and the float surface mode.

A surface of the fixed-surface mode has a fixed position in space. Most surfaces are used in this mode. They can be further divided into external surface mode and internal surface mode (see Chapter 2).

A surface in the periodic-surface mode has no fixed position in space. These surfaces are used to provide periodic boundary conditions. A surface in such a mode must be of an implicit type; that is, periodic boundary conditions have to be provided through analytic functions.

A surface in the float surface mode is an internal surface without location constraints. It is not a surface in the conventional sense. It is only a convenient way of grouping block faces, so that clustering can be done for them.

7.2 Fixed-surfaces – Implicit types

Under the fixed surface mode, an implicit surface is a surface specified by an equation in the form,

$$u(x, y, z) = 0$$

In general, $u(x, y, z)$ may be any function that can be programmed in C and satisfies the following conditions:

1) $u(x, y, z)$ is defined in a neighbourhood of the surface $u(x, y, z) = 0$. The neighbourhood should cover a domain larger than that expected for the initial positions of the corners which are assigned for the surface.

2) At any point in the intended physical domain, $u(x, y, z)$ should have a well defined gradient vector pointing to or away from the surface. The dependency of the gradient vector on (x, y, z) should be smooth.

3) The gradient vector of $u(x, y, z)$ should point into the region to be gridded.

7.2.1 Built-in implicit surfaces

Certain simple forms of implicit surfaces are built in with GridPro. They are:

The plane surface (`-plane`)

Example surface definition statement:

```
s  3  -plane  (0.1  0.2  0.3  0.4);
```

This statement defines surface 3 as a plane surface specified by the equation $0.1x + 0.2y + 0.3z + 0.4 = 0$ with the surface normal vector given by $(0.1, 0.2, 0.3)$. Notice that the parameter values to specify a given plane surface are not unique. They can be scaled by any fixed positive or negative constant without changing the surface. However, a negative scaling changes the direction of the normal vector (i.e. the orientation) of the plane.

The (super) ellipsoid surface (`-ellip`)

Example surface definition statement:

s 3 -ellip (0.1 0.2 0.3 4.0);

This statement defines surface 3 as a super ellipsoid specified by the equation $|0.1x|^4 + |0.2y|^4 + |0.3z|^4 - 1 = 0$. The gradient of the surface function $u(x, y, z) = |0.1x|^4 + |0.2y|^4 + |0.3z|^4 - 1$ is pointed to the outside of the super ellipsoid. The user must supply an orientation change operator '-o' when the gridded region is inside the ellipsoid (i.e. when the ellipsoid is an outer boundary). For a regular ellipsoid, the value for the power parameter is 2 (instead of 4 in the above example). The statement can be

s 3 -ellip (0.1 0.2 0.3 2.0);
or
s 3 -ellip (0.1 0.2 0.3);

7.2.2 Non-builtin implicit surfaces

A non-builtin implicit surface can be specified in the form of a '.h' file with the syntax of the C preprocessor. The up side of this is the flexibility of using different functional forms; The down side is that extra steps must be taken to compile a user generated function library in C and link it to the main module of GridPro before running GridPro.

Example surface definition statement:

s 3 -implic "torus.h";

This statement defines surface 3 to be an implicit surface as specified in the file "torus.h".

Let us say that we want "torus.h" to define a torus as follows: The central circle of the torus is on the y-z plane and is centered at origin with a radius 1.5. The cross-sectional circle of the torus has a radius of 0.5. The region to be gridded is inside the torus. The file "torus.h" is listed below,

Program 7.1 *File "torus.h"*

```
#define FUNCU ya=sqrt(y*y+z*z)-1.5, \
          1.0 - (ya*ya + x*x)*4  /* Define torus surface*/
#define Ulen    -1.0             /* -1.0 for no period  */
```

For those who do not have experience in the C programming language, three comments are in order: 1) A matching pair of '`/*`' and '`*/`' with everything in between will be ignored by the C compiler. They can therefore be used to make comments. 2) A '\' character at a line end continues this line to the next. 3) Each '`#define`' line defines a macro with the name of the macro and the content of the macro following.

If you want to define a different surface, you can copy this file and change the contents of the two pre-named macros, `FUNCU` and `Ulen`.

For `Ulen`>0, '`FUNCU` modulo `Ulen` = 0' specifies the surface; Otherwise '`FUNCU` = 0' specifies the surface.

In defining these macros, (x, y, z) are the input variables. Also available are 11 sets of auxiliary variables, (xa, ya, za), $(x0, y0, z0)$, .. , $(x9, y9, z9)$. They can be used to construct a surface function. These intermediate variables provide the opportunity to segment the surface formulation and to, thereby, achieve clarity. The segments are separated by the "," operator.

7.3 Fixed-surfaces – Explicit types

7.3.1 Surfaces of quadrilateral elements

Single patch bilinear parametric surface (`-linear`)

Example surface definition statement:

s 3 -linear "surf1.dat" ***[boundary_conditions]*** **;**

This defines a local bilinear parametric surface with certain *[boundary_conditions]*. The surface is specified by an IxJ array of grid points for some I and J in the file "surf1.dat". The bilinear interpolation is used to determine the surface points within the array cells. No collapsed cells or merged data points are allowed.

`The data format in file "surf1.dat"`: The first line of the data file lists I and J. The subsequent lines list the three coordinates of a surface grid point in the order of increasing J, then I. In terms of FORTRAN, the data can be read by a program as follows:

```
READ(UNIT,*) IMAX,JMAX
DO 10 I=1,IMAX
DO 10 J=1,JMAX
10   READ(UNIT,*) X(I,J),Y(I,J),Z(I,J)
```

The *[boundary_conditions]* in the example line specifies how the boundaries of the surface piece should be treated by GridPro. The options are:

`+i` – i=0 link to i=0 side
`-i` – periodic in I direction
`+I` – i=IMAX link to i=IMAX side
`+j` – j=0 link to j=0 side
`-j` – periodic in surface J direction
`+J` – j=JMAX link to j=JMAX side

At most, one of the `+i,-i,+I` options and one of the `+j,-j,+J` options can be used for a surface definition statement.

Instead of the inline *[boundary_conditions]*, a connectivity file named `surf1.dat.conn` can exist to provide connectivity information. In any case, if there is no inline *[boundary_conditions]* or there is the file `surf1.dat.conn`, the surface is treated as a special case of multi-patch surfaces, discussed in the next subsection.

The natural surface orientation is defined by the normal vector $\hat{i} \times \hat{j}$ where $\hat{i}$ and $\hat{j}$ are the unit vectors along the i index and j index directions respectively for any of the array points that determine the surface. If the natural orientation is not the same as expected (i.e. the positive side of the surface should be facing the region to be gridded), the reversing orientation flag `-o` can be placed in the surface definition statement as follows,

s 3 -linear "surf1.dat" ***[boundary_conditions]*** `-o` **;**

Note that, a surface here is specified by a grid which, however, is not the same as the surface grid generated with GridPro. The relation between them is that the generated surface grid will be on the surface which is specified by yet a different grid.

Multiple patch bilinear parametric surfaces (`-linear`)

A surface can be specified by multiple patches. The type for it is again `-linear`. Two files are involved to specify a surface of this type; namely, a data file which contains a sequence of surface patches, and a connectivity file which provides the information as to how the patches in the data file are connected. The format for each of the patches in the data file is the same as the single patch surface.

The requirements on the connections between different surface patches are very mild. They only need to be nearly-full face matchings. To understand it, you can imagine that a surface is composed of full face matching quadrilateral patches. However, the data specifying the surface on a patch does not need to exactly cover that patch. There can be gaps between adjacent surface patches. Preferably, the gap scales are smaller than the element (cell) scale of the surface patches near the gaps. However, no overlaps are allowed.

Example surface definition statement:

s 3 -linear "surf1.dat";

File "`surf1.dat`" is also called the data file. An associated connectivity file that has the name "`surf1.dat.conn`" may or may not exist. If "`surf1.dat.conn`" does not exist, GridPro will use the surface data to generate it by some default rules.

"`surf1.dat.conn`" contains the information how the surface pieces are connected. An example of the connectivity file consists of the following lines,

```
1   6  1   6  -1   2  2   0  0
2   3  1   3  -1   1  2   0  0
3   2  1   2  -1   4  2   0  0
4   5  1   5  -1   3  2   0  0
5   4  1   4  -1   6  2   0  0
6   1  1   1  -1   5  2   0  0
```

Each line here defines a surface piece and its connection to other pieces. It has the following format,

sp_id nxi sdi nxI sdI nxj sdj nxJ sdJ

sp_id – An id number that labels the surface piece. It should appear sequentially and correspond to the data piece in the data file.

nxi sdi – Boundary condition for the i = IMIN side of current surface piece. *nxi* provides the neighboring surface piece id. *nxi=-1* means no neighbor surface piece. *sdi* indicates which side of the neighbor surface piece is connecting to the i = IMIN side of current surface piece. Values *sdi=1,-1,2*, and *-2* mean the i=IMIN side, i=IMAX side, j=JMIN side, and j=JMAX side respectively. If *nxi*$\leq$*0*, the value of *sdi* is irrelevant.

nxj sdj – Boundary condition for the j = JMIN side of current surface piece.

nxI sdI – Boundary condition for the i = IMAX side of current surface piece.

nxJ sdJ – Boundary condition for the j = JMAX side of current surface piece.

The orientation of the surface will be synchronized to that of the first piece.

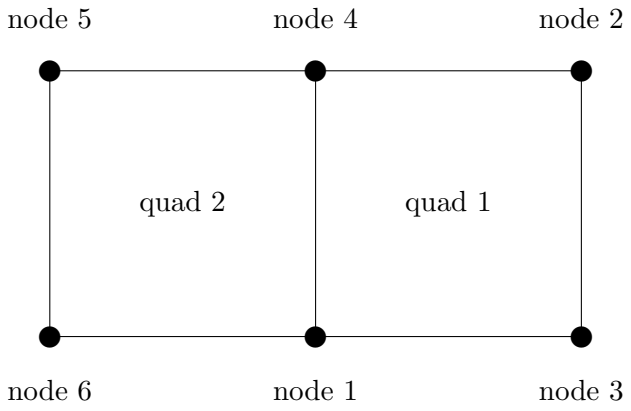


Figure 7.1: An example of a surface of quads with two quads.

Surfaces of unstructured quad elements

A surface point is specified by bi-linearly interpolating on the quad elements.

Example surface definition statement:

s 3 -quad "surf1.dat";

GridPro will interpret the data format according to the contents of the data file. It may be in one of the two implemented formats. The first is the small field Nastran bulk data fixed or free format, where the GRID entry is used to define a node coordinate position, and the CQUAD4 or CQUADR entry is used to specify a quad element (for detail see MSC/NASTRAN Quick Reference Guide).

The implementation is a subset of MSC/NASTRAN's specification. In particular, the replication capability is not implemented. That is, data entries should not contain the '=' and '*' operators. For the free format, GridPro requires that data entries are separated by commas.

The second data format is the GridPro format used by GridPro. As an example, '`surf1.dat`' may consist of the following lines:

```
6
801.97693   -7.0479345   -23.675423
886.42688    14.096904    -15.55965
942.72565    28.191708   -10.122647
999.00952    42.283035   -4.6956768
1055.3088    56.378807   0.71006197
1111.5261    70.471634    5.0577559
2
1 3 2 4 0
6 5 4 1 0
```

It is a node list followed by a quad list. The first line specifies that there are 6 nodes in this file. It is followed by the coordinates (x y z) of the 6 nodes (one node per line). The next line indicates that there are 2 quads in the data. Then, each of the following lines specifies the 4 node numbers and 1 property id for that quad. The node numbers are ranging from 1 to 6 corresponding to the nodes in the node list. In this case, quad number 1 is formed by nodes 1, 3, 2, and 4, and quad number 2 is formed by nodes 6, 5, 4, and 1. The property id is not used

in the graphical manager and Ggrid. It is used in some utility calculations. Therefore, the value of property id can be arbitrary here.

The listing order of node ids for a quad should be circling about the quad in either of the two possible directions. No more than one of the 4 sides of a quad may be collapsed either by having the same node id or by having a single space position. Small holes are allowed on the surface.

The natural orientation of a quad surface is determined by the first quad in the quad list. The positive side of the surface is the side where when one faces the first quad, the listing order of nodes that defines the quad rotates anti-clock wise (right hand rule).

7.3.2 Surfaces of triangular elements (`-tria`)

There is one type in this category, namely '`-tria`'. The data formats are similar to that of unstructured quad surfaces. Mainly, the differences will be pointed out here.

1) The Nastran data format entries used here are GRID, CTRIA3, and CTRIAR.

2) In the GridPro format, the element is defined by 3 nodes.

Example surface definition statement:

s 3 -tria "surf1.dat";

In the GridPro format, 'surf1.dat' may consist of the following lines:

```
4
886.42688     14.096904     -15.55965
942.72565     28.191708    -10.122647
999.00952     42.283035    -4.6956768
1055.3088     56.378807    0.71006197
2
1 3 2 0
4 1 2 0
```

Here we have 4 nodes and 2 triangles. No degenerate sides are allowed for any of the triangular elements.

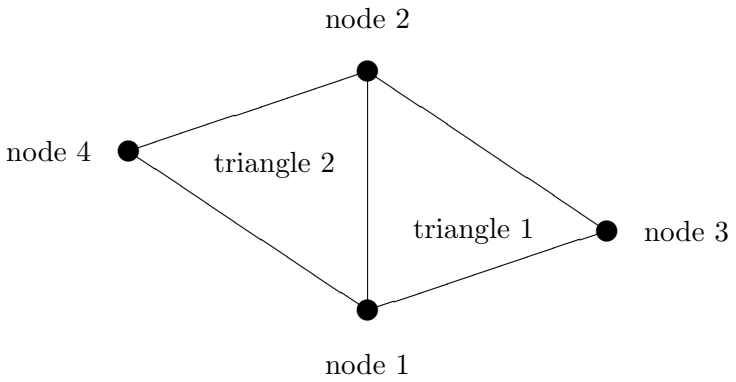


Figure 7.2: An example of a surface of triangles with two elements.

7.3.3 Surfaces of revolution (`-tube`)

A surface of this type is specified by the revolution around a digitized center curve. Therefore, it is a more general type of surface of revolution.

Example surface definition statement:

s 3 -tube "surf1.dat";

Here 'surf1.dat' contains the data for the digitized center curve. It can be read via,

```
READ(UNIT,*) IMAX
DO 10 I=1,IMAX
10  READ(UNIT,*) X(I),Y(I),Z(I),R(I)
```

Where `X(I),Y(I),Z(I)` are the coordinates of the $\texttt{I}^{th}$ center curve data point and `R(I)` is the radius of revolution for the data point.

The center curve point, the center curve tangent and the the radius of revolution are linearly interpolated from the center curve data.

The revolution for each point on the center curve is performed in the normal plane of that point with respect to the tangent of the curve. Thus, this type represents tube - like surfaces with curved center line and variable radius.

To avoid ill-specified surfaces, the circular disks bounded by the circle generated by the revolution for any two center curve data points should not intersect each other.

The natural orientation is pointed to the outside of the tube section defined by the first two data points on the center curve. Therefore the first two data points should not have the same position in space. However, in general two center curve data points *can* have the same position in space, as long as the radii are different.

An interesting example is a torus surface. There are two ways to represent it with surface type '`-tube`'. The first is to digitize the center circle of the torus, and use a constant radius for all of the center curve data points. The second way is to digitize the axis of rotational symmetry, and use a variable radius to represent the cross-sectional circle of the torus.

For both cases, the '`-i`' flag must be given to the surface definition statements to indicate that the center curves have a periodic boundary condition on them.

7.4 Periodic surfaces

A periodic surface is a surface used in the periodic mode; that is, it is used to determine a periodic boundary condition for the grid to be generated. The word "periodic" here should not be confused with the the word "periodic" used to indicate the periodic boundary condition for a surface. A surface may have loops that are closed with such conditions (instead of, for the grid to be generated).

A surface used to specify a periodic boundary condition has different and stronger requirements on the surface functions. More precisely, a periodic boundary condition is specified by providing a non-singular coordinate transformation in the form,

$$\begin{aligned} U &= u(x, y, z) \\ V &= v(x, y, z) \\ W &= w(x, y, z) \end{aligned}$$

A surface with the given periodic boundary condition is selected by GridPro among all the surfaces satisfying the condition,

$$\begin{array}{rcrcc} u(x,y,z) & - & u(x_{id},y_{id},z_{id}) & = & period \\ v(x,y,z) & - & v(x_{id},y_{id},z_{id}) & = & 0 \\ w(x,y,z) & - & w(x_{id},y_{id},z_{id}) & = & 0 \end{array}$$

on every pair of grid points (x, y, z) and (x_{id}, y_{id}, z_{id}), for which the identifying relation is defined by `i`-statements in the TIL program.

7.4.1 General implicit surfaces (`-implic`)

Similar to a non-builtin implicit surface in the fixed surface mode, an implicit surface in the periodic surface mode can be specified in the form of a '`.h`' file with C control-line syntax. In terms of their appearance as `s`-statements, there is no difference, except the period must be provided on the surface definition line in the TIL code. For example,

s 3 -implic "polar_for_z.h" 30.0;

where `30.0` indicates the period.

However, in the '`.h`' file, 9 pre-named macros are used.

`FUNCU, Ulen, FUNCV, Vlen, FUNCW,` and `Wlen` are used to define the forward transformation from (x, y, z) to (u, v, w) and `FUNCX, FUNCY,` and `FUNCZ` are used to define the inverse transformation from (u, v, w) to (x, y, z). At run time, the consistency of the inversion is checked using the initial positions of all the grid points assigned to the involved surface.

The way to define `FUNCU, Ulen, FUNCV, Vlen, FUNCW,` and `Wlen` is the same as for a non-builtin implicit surface in the fixed surface mode. However, to define `FUNCX, FUNCY,` and `FUNCZ`, one needs to use (u, v, w) as input variables instead of (x, y, z). For the details, see the next subsubsection.

7.4.2 The polar periodic BC (`-xpolar`)

One of the hard wired implicit surfaces is for the periodic boundary condition on the angular coordinate in the polar coordinate system with x-axis as the rotation axis. The angle is measured in degrees.

Example surface definition statement:

s 3 -xpolar 30.0;

where `30.0` indicates the period is 30 degrees. There is no other data specification needed.

The transformation used is,

$$\begin{array}{rcc} U & = & \mathrm{atan}(\frac{z}{y}) \cdot 180/\pi \\ V & = & x \\ W & = & \sqrt{z^2+y^2} \end{array}$$

Two points identified by this periodic surface condition will have the same V and W values and a fixed difference in U.

The corresponding '`.h`' file would appear as,

Program 7.2 *File* 'period.h'

```
#define FUNCU ((180/PI)*atan(z/(y+dsig(y)*1.0e-30))+((y<0)? 180 : 0))
#define Ulen  ( 360 )

#define FUNCV  x
#define Vlen  (-1.0)

#define FUNCW  sqrt(z*z+y*y)
#define Wlen   (-1.0)

#define FUNCX   v
#define FUNCY  (w*cos(u*PI/180))
#define FUNCZ  (w*sin(u*PI/180))
```

Here, certain things are added in to prevent an over flow condition. `PI` and `PI2` are predefined. At the end of `FUNCU`, '`((y<0)? 180 : 0))`' means that if $y < 0$ then the value is 180; otherwise it is 0. This is just some C programming syntax.

Rotation and translation operators can be applied in the s-statement to change the rotation axis of the polar system. For the polar system with z-axis as the rotation axis, one has,

```
s 3 -xpolar 30 -R 0 0 1   0 1 0   1 0 0;
```

7.4.3 The cartesian periodic BC (-xyz)

The other hard wired implicit surface is for the periodic boundary condition on the x-axis in the cartesian coordinate system.

Example surface definition statement:

s 3 -xyz 2.5;

where `2.5` indicates the period is 2.5. There is no other data specification needed.

The transformation used is,

$$\begin{aligned} U &= x \\ V &= y \\ W &= z \end{aligned}$$

Two points identified by this periodic surface condition will have the same V and W values and a fixed difference in U (hence x).

The corresponding '`.h`' file would appear as,

Program 7.3 *File* 'period.h'

```
#define FUNCU (x)
#define Ulen  (-1.0 )

#define FUNCV (y)
#define Vlen  (-1.0)

#define FUNCW (z)
#define Wlen  (-1.0)
```

```
#define FUNCX  u
#define FUNCY  v
#define FUNCZ  w
```

Rotation and translation operators can be applied in the s-statement to change the rotation axis of the polar system. For the cartesian system with z-axis as the periodic axis, one has,

```
s 3 -xyz 2.5 -R 0 0 1   0 1 0   1 0 0;
```

7.5 Float surfaces

As we said earlier, the purpose of float surfaces is to provide a convenient way of grouping block faces, so that clustering can be applied for them. Unlike surfaces in other modes, there is no location constraints on a float surface. An example statement is:

s 1 -float +c 0.01;

Here the float mode is indicated by the type parameter `-float`, and the average spacing on either side of the surface is 0.01. Other than that, there is no location constraint and it can be internal to the topology (that is, blocks can be on both sides of the surface), the use of a float surface is very similar to that of a fixed surface. In particular, corners can be assigned to the surface; And the surface can be added to or deleted from a corner, an edge, or a face. The clustering is turned on or off along with the corresponding clustering group. However, when one assigns corners or other objects to a float surface, they must satisfy a consistent requirement. That is, all the block faces on a float surface must be in the same face sheet. In turn, a face sheet is defined as a side of a block sheet. Note that one can have a part of a face sheet assigned to a float surface.

Clustering may also be applied for only one side of the surface. A statement such as,

s 1 -float 0 +c 0.01;

will apply clustering for the side 0. The other side is side 1. The determination of side 0 or 1 is best done by experiment.

7.6 Surface transformations

In the TIL code, a surface can be linearly transformed and translated with the corresponding option flags and arguments in the s-statement. The vector expression form of such transformation is explained in Section 2.4 of Chapter 5.

7.7 Surface conditions

In this section, we will discuss certain requirements that GridPro imposes on the surfaces.

7.7.1 Smoothness

A surface used by GridPro must be "smooth" . However, whether a given surface is smooth or not strongly depends on the topology you choose for the blocks on that surface.

To illustrate the concept of smoothness here, let's use a simple example. Consider an airfoil with a sharp trailing edge. Suppose that the airfoil is specified by a surface of type `-linear` with a dense enough supply of points. Then, most of the places on the surface are smooth for GridPro. However at the sharp trailing edge, when one goes from one surface data point to another, the surface tangent vector will turn by a very large amount (close to 180°).

For the case where a C-cut topology is used, a block boundary is forced to go to this point. Therefore, the sharp trailing edge will not cause a problem and the surface will be regarded as smooth.

On the other hand, if an O-type topology is chosen, GridPro will automatically distribute grid points taking the airfoil as a seamless whole. In this case, the surface point on the sharp trailing edge will be treated the same as any other surface point and the large tangent turn will be regarded as non-smooth.

For a non-smooth surface, it can be restructured by adding surface data points to round off those large tangent turns. In doing so, the surface shape may be changed slightly, but since it can be done at a very fine scale, the changes should not affect the flow field solutions.

One should also avoid specifying a surface with too many data points since it wastes both memory and CPU time. In terms of tangent turning rates, limiting to less than 1° turns will give very good surface specifications, even though GridPro can handle tangent turns as large as 90°, provided that they do not form clusters or sharp points.

7.7.2 Intersections

In general, if two surfaces are supposed to intersect, the surface specifications should be provided to extend somewhat beyond the intersection. GridPro will determine the actual intersection automatically once the extensions are in place.

Part II

Appendices

Appendix A

Quick Reference to Schedule Syntax

A schedule file consists of two sections: A mandatory schedule section and an optional output section. A line is continued to the next by ending with a '\' character.

Conventions: When a *num* is used to refer to a corner or surface, it represents an internal corner or surface id, which can be obtained through running GridPro in the debug mode or with a proper `PRINT..` statement in the topology file. A *label* is a name string defined in the TIL program for a collection of objects of the same type. An object is either a corner, an edge, a face, a block or a surface.

A.1 Schedule section

The schedule section is composed of a sequence of `step`s with the syntax as follows:

`step` *num*: *actions*

where *num* is a step label and *actions* is a sequence of actions that will be executed from left to right. `Step` is a convenient way to group actions to be executed. The steps are executed one by one. If a step is in the gap of the steps listed in the schedule file it is implied that the actions for this step are the same as those of the next nearest step explicitly specified in the file. The syntax of most actions has two basic structures:

1) *–flag* {*obj_list*} {*parameters*} and

2) *–flag obj parameter obj parameter ...* .

The possible actions for a step are:

`-g` ***label num...***
— Change the grid density on edges collected in *label* to *num*. If *num*is 0, the edges with the *label* are redesignated as default edges and assigned the default grid density.

`-g` ***label d*x ...**
— Change the grid density for edges collected in *label* to *d*times the current density rounding off to the integer part.

For the `-g` action, *label* can be predefined words 'ALL' or 'all' to mean all edges, and 'DEF' or 'def' to mean the edges with default grid density (default edges).

`-a` ***label num...***
— Accelerate the convergence of blocks collected in *label* with a loop count *num*.

`-a` ***num1 num2 ...***

— Accelerate the convergence of block *num*1 with a loop count *num*2.

`-S` ***{num}***

— {Set the length of sweep laps to *num*, and} run GridPro for one lap of sweeps.

`-C` ***num***

—change the internal clustering resetting interval to *num* sweeps. That is, the internal cluster parameters will be re–evaluated every *num* sweeps. *num* must be >= 1. The default value is 5.

`-C` ***surf_list d1 d2***

— Setting clustering parameters for surfaces. *surf_list* is a list of surface items separated by one or more spaces. A surface item is either a surface label or a surface range bounded by the internal surface ids. *d*1 and *d*2 are two clustering parameters for the listed surfaces.

Affected Surfaces: Not all surfaces in the list are affected. The rule is that, if at least one of the listed surfaces has its spacing parameter set in the TIL code, then only surfaces with their spacing parameters set in the TIL code will be affected, otherwise all the surfaces in the list will be affected.

About *d*1: If the affected surfaces have the spacings specified, *d*1 is a scaling factor to the spacings, that is, the target spacing for a surface will be `specified_spacing`**d*1. Otherwise, *d*1 is the targeted spacing ratio for the affected surfaces.

About *d*2: *d*2 gives the grid range used for clustering. At most, one layer of block from the surface can be used for clustering. Let K be the number of grid layers in the block. If $d2 < 1$, the number of grid layers affected is K**d*2. Otherwise, the number of grid layers affected is `min`{*d*2,K}.

Turning off Clustering: If either of *d*1 or *d*2 is <= 0, the clustering is turned off for the affected surfaces.

`-c` ***num1*** [***num2***]

—change the algebraic clustering multiple and algorithm.

About *num*1: make the new cell count to to be a multiple of *num*1. *num*1 must be >= 1. The default is 1.

About *num*2: select the algorithm *num*2. *num*1 must be 0, 1, 2, or 3. The default is 2.

0 – linear algorithm with average spacings.

1 – curve fit algorithm with average spacings.

2 – curve fit algorithm with equalized spacings.

3 – linear algorithm with equalized spacings.

`-c` ***surf_list d1 d2***

— Setting the algebraic clustering parameters for surfaces. *surf_list* is a list of surface items separated by one or more spaces. A surface item is either a surface label or a surface range bounded by two numbers that represent two internal surface ids. *d*1 and *d*2 are two clustering parameters for the listed surfaces.

Affected Surfaces: Only surfaces with their spacing parameters set in the TIL code will be affected.

About *d*1: $d1$ is a scaling factor to the spacings specified in the TIL code, that is, the target spacing for a surface will be `specified_spacing`$*d1$.

About *d*2: $d2$ gives the cell growth ratio for clustering. $d2$ must be > 1.0 and < 2.5. The default is 1.5.

`-w` **{*num*}**
— If *num* > 0, set the output interval to *num* sweeps. Without *num* , the output interval is unchanged, but output is done once immediately. What to output is determined in the output section of the same schedule file. If `-w ..` is an action before any sweep is run, action "`-w`" will output the initial setup, and action "`-w -1`" will output the initial setup with the surface grid points projected on surfaces.

`-r` **{*num*}**
— Readjust surfaces with radius = *num*. Without *num*, the default value for radius is 1.0; The affected surfaces are those marked with `-r` flag in the TIL code.

`-r` ***surf_list num***
— Readjust surfaces with radius = *num*. The surfaces in *surf_list* are readjusted. *surf_list* is a list of surface items separated by one or more spaces. A surface item is either a surface label or a surface range bounded by the internal surface ids (e.g. 2..5 SURF1).

`-scpl` ***surf_list d***
— Set the surface–volume coupling constants for surfaces in *surf_list* to *d*. *d* should be greater than 0 (default = 1.0).

`-s`
— Switch to the script control mode in which actions are read in from the schedule file.

`-m`
— Switch to on-line control mode in which actions are typed in from the keyboard.

`-v` ***num d***
— Set the volume relaxation count per sweep to *num* and set relaxation constant to *d*.

`-sys` "script with args"
— Run a Unix or PC script file (for post processing grid).

`-D`
— Display current run-parameter settings.

`-R` ***parameter value***
— Change the value of run-parameter *parameter* to *value*. For a list of settable *parameter*s, use the `-D` action. Some of the most used parameters are:

`CTRL.SINGULAR` ***d*** – *d* is a number between 1.0 and 2.0. for most cases 1.25 is an adequate choice.

`CTRL.CURVA.STRENGTH` ***d*** – for curvature control. *d* should be in the range [0,2]. This parameter provides an *average* importance of curvature contribution relative to the other grid quality measures in the grid generation.

`CTRL.CURVA.LIMIT` ***d*** – for curvature control. *d* should be in the range [0,2]. This parameter is a supplement to the parameter above. The `STRENGTH` parameter is specified in an average sense. At the local level, the `STRENGTH` is limited by the `LIMIT` parameter to eliminate possible contribution spikes.

`CTRL.CURVA.CUTOFF.LOWER` ***d*** – for curvature control. *d* should be in the range [0,90] and less than `CTRL.CURVA.CUTOFF.UPPER`. *d* provides a relative curvature threshold below which the normalized relative curvature is regarded as 0. Therefore, a larger value makes the distribution less sensitive to the curvature.

`CTRL.CURVA.CUTOFF.UPPER` ***d*** – for curvature control. *d* should be in the range [0,90] and greater than `CTRL.CURVA.CUTOFF.LOWER`. *d* provides a relative curvature threshold above which the normalized relative curvature is regarded as 90 degrees. Therefore, a larger value makes the distribution less sensitive to curvature.

`CTRL.SPACE.STRENGTH` ***d*** – for spacing ratio control. *d* should be in the range [0,2]. This parameter provides an *average* importance of grid smoothness contribution relative to the other grid quality measures in the grid generation.

`CTRL.SPACE.LIMIT` ***d*** – for spacing ratio control. *d* should be in the range [0,2]. This parameter is a supplement to the parameter above. The `STRENGTH` parameter is specified in an average sense. At the local level the `STRENGTH` is limited by the `LIMIT` parameter to eliminate possible contribution spikes.

`CTRL.SPACE.CUTOFF.LOWER` ***d*** – for spacing ratio control. *d* should be in the range [0,1] and greater than `CTRL.SPACE.CUTOFF.UPPER`. This gives the inverse of sensitivity level to the spacing ratio. *d* provides a relative threshold below which the spacing ratio is regarded as 1.0.

`CTRL.SPACE.CUTOFF.UPPER` ***d*** – for spacing ratio control. *d* should be in the range [0,1] and less than `CTRL.SPACE.CUTOFF.LOWER`. This gives the inverse of sensitivity level to the spacing ratio. *d* provides a relative threshold below which the spacing ratio is regarded as 1.0.

NOTE: Schedule file can be modified during a run to steer the run.

A.2 Output section

The output section determines what to output when a '`-w`' action is encountered in the schedule section. The output section is composed by lines beginning with the key word '`write`'. The syntax is:

`write` *which* {*what*} {*where*}

which = Specify which block(s) to write out. There are four choices:
- `-b` *bid* – the bid^{th} block.
- `-c` *cid1 cid2* – the block defined by corners *cid1* and *cid2*.
- `-a` – all the blocks

what = Specify what kind of data to write out. It is a combination of the following flags:
- `-D` *dim* – dimension parameter:
  0 is for binary dump for restart.
  2 is for 2-d grids for 2-d runs.

3 is for 3-d grids.

`-d` – output the dual grids.

`-m` *cid1 cid2*

Or `-m` *dir* – Output a maximum chain of blocks starting with the one defined in *which* and chaining in the direction defined by Face(*cid1*, *cid2*), or by the direction *dir*(=0..5)

The default is 3-d grids.

where = Specify where to output the block(s). There are two choices:

`-f` *fn* – output will over-write file *fn*.

`-F` *fn* – output will be appended to file *fn*.

The default file is '`dump.tmp`' for dumping and '`blk.tmp`' for others.

Appendix B

Worked Out Examples

B.1 Operational aspects

B.1.1 Use only an editor

(1) Sketch topology

(2) Label the sketch

(3) Translate sketch into TIL code

(4) Put TIL in "name.fra"

(5) Make a schedule

(6) Put schedule in "name.sch"

(7) Run case by the command
>Ggrid name.fra <ret>

B.1.2 Use the GridPro®/az3000 Graphic Manager

(1) Start manager with the command
>az <ret>

(2) Create topology

(3) Debug topology

(4) Launch run

(5) Examine grid

(6) Stop the run

(7) Topology is saved in "_az.fra"

B.1.3 Use both the Graphic Manager and an Editor

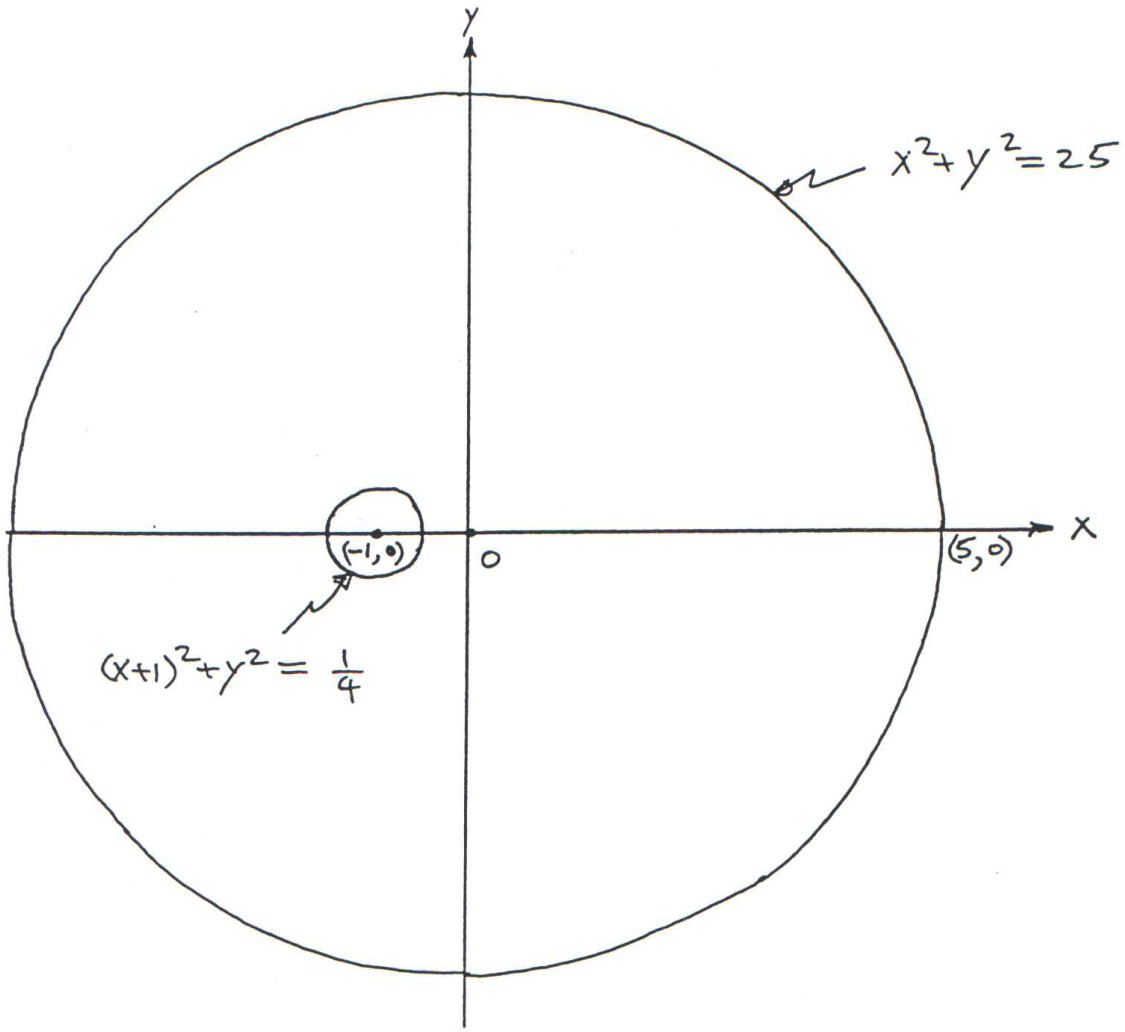
(1) Create or edit ".fra" files

(2) Run cases

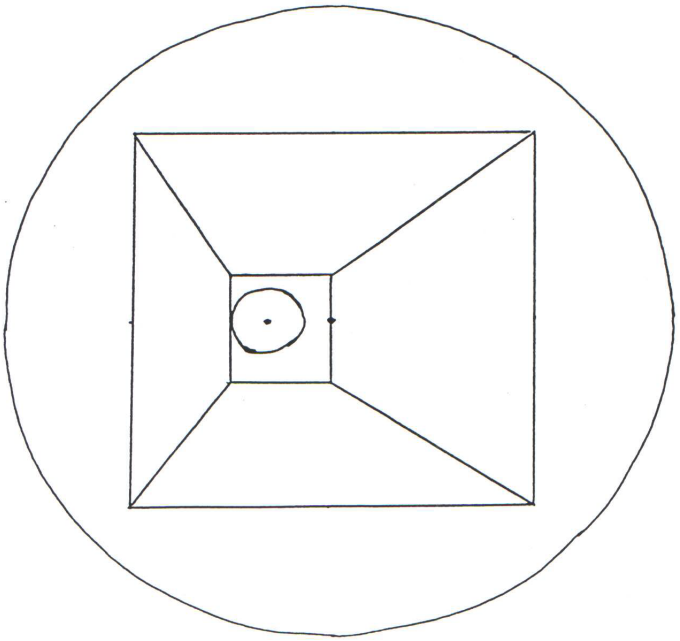
(3) Viewing grid
>az <ret>

B.2 Running GridPro®/az3OOO in /Cases/gridl:

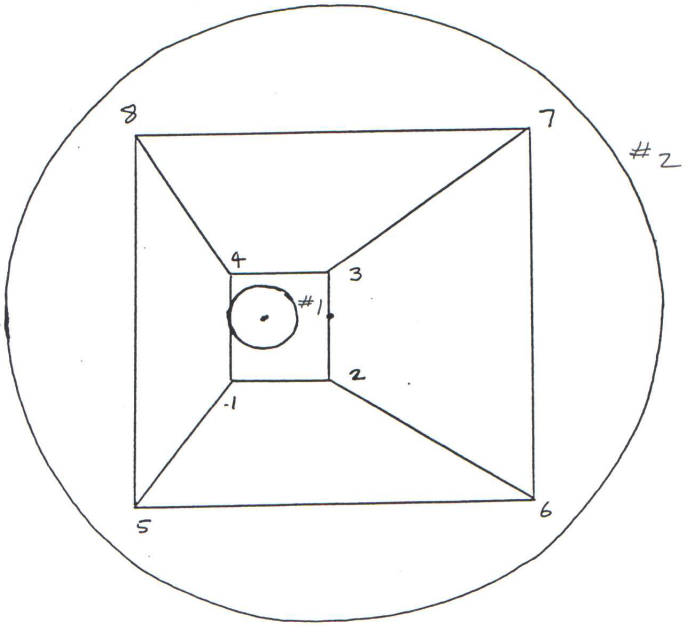
```
> cd    ~/Cases/grid 1   <ret>
> Ggrid example1.fra    <ret>   <ret>
```



To setup the pattern of points, we first sketch a box about the inner circle and a corresponding larger box inside the outer circle. Then we connect the corresponding corners. The result is a wire frame figure that is commonly called a "hypercube". The sketch appears below along with the two circular boundaries. Notice that no corner points were put on the boundaries.

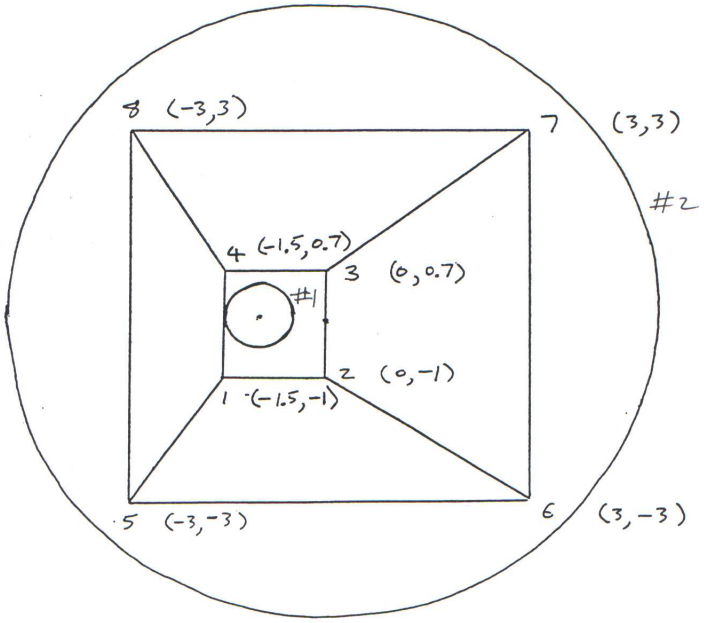


Now that we have sketched the figure and have drawn in a simple hypercube for a wire frame pattern, we must prepare our diagram for encryption into TIL code. This requires us to label the boundary surfaces and the corner points. The labelled sketch appears below



The labels in the above figure show the relationship between the corners and also the relationship between corners and the boundary surfaces. To complete the specification, it

remains to give the actual locations of the comers. The amended figure appears below.



B.3 A Two Dimensional Donut

B.3.1 Surfaces

```
(ax)  * (ax)  + (by)  * (by)  + (cz)  * (cz)  -1 = 0
(2*x) * (2*x) + (2*y) * (2*y) + (0*z) * (0*z) -1 = 0
(x/5) * (x/5) + (y/5) * (y/5) + (0*z) * (0*z) -1 = 0
```

Program B.1 *File: 'donut.fra'*

```
SET   DIMENSION   2
SET   GRIDDEN     6
COMPONENT   annular()
BEGIN
   s  1  -ellip(  2   2  0)  -t  -1  0  0 ;  #inner circle
   s  2  -ellip(0.2 0.2  0)  -o ;  #outer circle
   c  1  -1.5   -1   0   -s  1  ;
   c  2     0   -1   0   -s  1   -L  1  ;
   c  3     0  0.7   0   -s  1   -L  2  ;
   c  4  -1.5  0.7   0   -s  1   -L  3 1;
   c  5    -3   -3   0   -s  2   -L  1  ;
   c  6     3   -3   0   -s  2   -L  2 5;
   c  7     3    3   0   -s  2   -L  3 6;
   c  8    -3    3   0   -s  2   -L  4 7;
   g  1  5  16  ;
   x  f  1  3   5  7  ;
END
```

Program B.2 *File: 'donut.sch'*

```
step 1:  -S  100  -w
write  -a  -f  blk.tmp
write  -a  -D  0  -f   dump.tmp
```

To generate the grid, type

Ggrid donut.fra <return>

To run GridPro™/az3000, we create a working directory and within which we put donut.fra and donut.sch. Next, we make sure that the environment variable GRIDPRO is set to "/usr/local/gridlib" or wherever the installation is. This can be checked with the command

setenv 1 grep GRIDPRO <return>

If it is not properly set, then it can be taken care of with the command

setenv GRIDPRO dir_path <return>

This line can be added to your ".login" file in your home directory. To generate the grid we then type in the command

Ggrid donut.fra <return>

The output data appears in the file "blk.tmp" in a 3D format. In terms of FORTRAN, the results can be read in the format

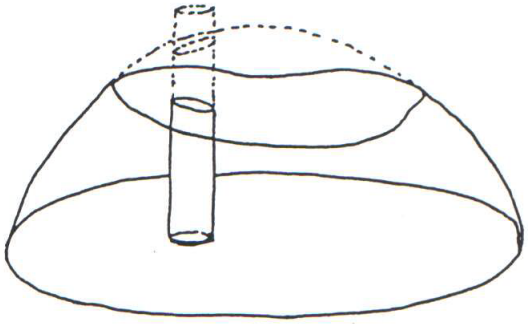
Program B.3 *Point data format in FORTRAN*

```
READ(UNIT,*) IMAX,JMAX,KMAX
DO 10 I=1,IMAX
DO 10 J=1,JMAX
DO 10 K=1,KMAX
10  READ(UNIT,*) X(I,J,K), Y(I,J,K), Z(I,J,K)
```

B.4 A Rod through an Enclosure

B.4.1 Surfaces

$$(x+1)*(x+1) + y*y = 0.25$$
$$x*x + y*y + z*z = 25$$
$$(0.25x)*(0.25x) + (0.17y)*(0.17y) + (0.5*(z-5))*(0.5*(z-5)) = 1$$
$$0*x + 0*y + l*z + 0 = 0$$



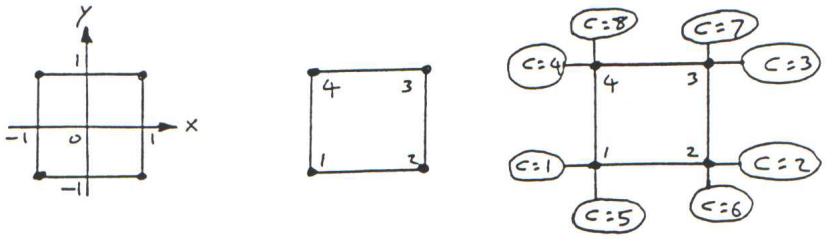
Program B.4 *File: 'RopSphereCap.fra'*

```
SET   GRIDDEN     8
COMPONENT   RodSphereCap()
BEGIN
   s  1  -ellip(   2    2    0)  -t  -1  0  0 ;  #rod
   s  2  -ellip( 0.2  0.2  0.2)  -o ;  #sphere
   s  3  -ellip(0.25 0.17  0.5)  -t  0  0  5 ;  #cap for sphere
   s  4  -plane(0       0    1    0) ;  #plane through equator

   c  1  -1.5   -1   0   -s  1  4  ;
   c  2     0   -1   0   -s  1  4  -L  1  ;
   c  3     0  0.7   0   -s  1  4  -L  2  ;
   c  4  -1.5  0.7   0   -s  1  4  -L  3 1;

   c  5    -3   -3   0   -s  2  4  -L  1  ;
   c  6     3   -3   0   -s  2  4  -L  2 5;
   c  7     3    3   0   -s  2  4  -L  3 6;
   c  8    -3    3   0   -s  2  4  -L  4 7 5;

   c  9  -1.5   -1  3.5  -s  1  3  -L  1  ;
   c  10    0   -1  3.5  -s  1  3  -L  2 9;
   c  11    0  0.7  3.5  -s  1  3  -L  3 10;
   c  12 -1.5  0.7  3.5  -s  1  3  -L  4 11 9;

   c  13   -3   -3  3.5  -s  2  3  -L  5 9;
```

```
   c  14    3   -3  3.5  -s  2  3  -L  6 10 13;
   c  15    3    3  3.5  -s  2  3  -L  7 11 14;
   c  16   -3    3  3.5  -s  2  3  -L  8 12 15 13;

   g  1  5  12  ;
   x  f  1 3   5  7   9  11   13  15  ;
END
```

Program B.5 *Sub-Program 1*

```
COMPONENT   loop4(SIN  s1,  s2,  cIN  c[1..8])
BEGIN
   c  1  -1   -1   0   -s  s1  s2  -L  c:1 c:5  ;
   c  2   1   -1   0   -s  s1  s2  -L  c:2 c:6  1  ;
   c  3   1    1   0   -s  s1  s2  -L  c:3 c:7  2  ;
   c  4  -1   -1   0   -s  s1  s2  -L  c:4 c:8  3  1  ;
END
```



Program B.6 *File: 'RopSphereCap.fra'*

```
COMPONENT   RodSphereCap()
BEGIN
   s  1  -ellip(   2    2    0)  -t  -1  0  0 ;  #rod
   s  2  -ellip( 0.2  0.2  0.2)  -o ;  #sphere
   s  3  -ellip(0.25 0.17  0.5)  -t  0  0  5 ;  #cap for sphere
   s  4  -plane(0       0    1    0) ;  #plane through equator

   INPUT 1 (-1  0  0) * (0.6  0  0   0  0.6  0   0  0  0.6) *
     loop4(sIN (1), (4), cIN (-8), cOUT (1..4));
   INPUT 2 (3  0  0   0  3  0   0  0  3) *
     loop4(sIN (2), (4), cIN (1:1..4 -4), cOUT (1..4));
   INPUT 3 (-1  0  3.5) * (0.6  0  0   0  0.6  0   0  0  0.6) *
     loop4(sIN (1), (3), cIN (1:1..4 -4), cOUT (1..4));
   INPUT 4 (0  0  3.5) * (3  0  0   0  3  0   0  0  3) *
     loop4(sIN (2), (3), cIN (2:1..4 3:1..4), cOUT (1..4));
```

```
   g  1:1  2:1  12 ;
   x  f  1:1 1:3   2:1 2:3   3:1 3:3   4:1 4:3;
END

COMPONENT   loop4(SIN  s1,  s2,  cIN  c[1..8])
BEGIN
   c  1  -1   -1   0   -s  s1  s2  -L  c:1 c:5  ;
   c  2   1   -1   0   -s  s1  s2  -L  c:2 c:6  1  ;
   c  3   1    1   0   -s  s1  s2  -L  c:3 c:7  2  ;
   c  4  -1   -1   0   -s  s1  s2  -L  c:4 c:8  3  1  ;
END
```

B.5 Exercises

1. Generate a grid with TIL Program B.1. This has the frame file donut.fra and the schedule file donut.sch. The schedule is for 100 sweeps with a write. Look at the grid with the viewer. Then rerun the case with 200 sweeps and look at the grid again. Move comer 2 from (0, -1) to the origin (0, 0) and rerun the case. Examine the results with the viewer. Are there any differences between the results with the extra 500 sweeps or with then new position for corner 2?

2. Run TIL Program B.4 and TIL Program B.6 with the schedule as of donut.sch for 100 sweeps. Repeat the runs to get a write at 200 sweeps. Examine the results for any possible differences. Modify either TIL Program B.4 or B.6 to lift the uppermost inner 4-loop from z=3.5 to z=4.0. Run the case again to see the results at 100 and then 200 sweeps. What are the differences if any?

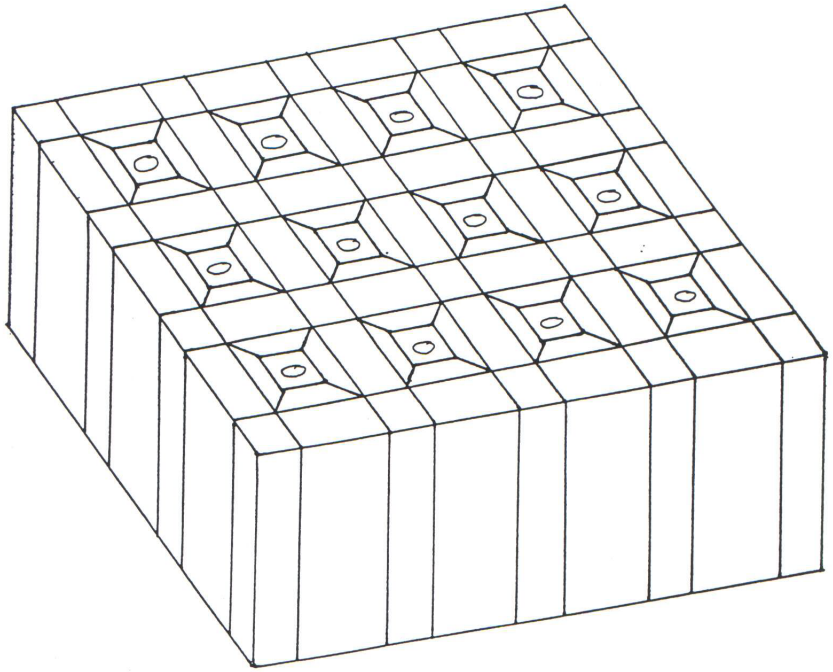
3. Modify TIL Program B.6 to remove the ellipsoidal cap. That is, just remove the cap and the assignments to it from the uppermost 4-loop. This gives us the simplifier case of a rod in a sphere. The TIL code for it is then given by

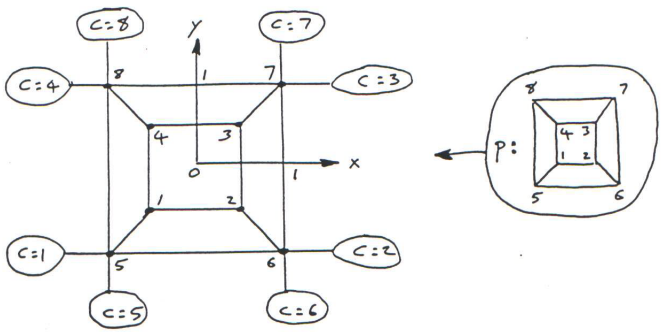
```
COMPONENT   RodSphereCap()
BEGIN
   s  1  -ellip(   2    2    0)  -t  -1  0  0 ;  #rod
   s  2  -ellip( 0.2  0.2  0.2)  -o ;  #sphere
   s  3  -plane(0       0    1    0) ;  #plane through equator
   INPUT 1 (-1  0  0) * (0.6  0  0   0  0.6  0   0  0  0.6) *
     loop4(sIN (1), (4), cIN (-8), cOUT (1..4));
   INPUT 2 (3  0  0   0  3  0   0  0  3) *
     loop4(sIN (2), (4), cIN (1:1..4 -4), cOUT (1..4));
   INPUT 3 (-1  0  3.5) * (0.6  0  0   0  0.6  0   0  0  0.6) *
     loop4(sIN (1), (3), cIN (1:1..4 -4), cOUT (1..4));
   INPUT 4 (0  0  3.5) * (3  0  0   0  3  0   0  0  3) *
     loop4(sIN (2), (3), cIN (2:1..4 3:1..4), cOUT (1..4));
   g  1:1  2:1  12 ;
   x  f  1:1 1:3   2:1 2:3   3:1 3:3   4:1 4:3;
END
```

Put this into a frame file rod_thru_sphere.fra and form a schedule file by putting the contents of donut.sch into rod_thru_sphere.sch. Then generate the grid with 100 sweeps. What is your assessment of the result? Continue by running the case further, for example to 200 sweeps. Is there any distinctive difference. What is happening?

4. In the example of program B.6, an improvement can be obtained from a change of topology. While keeping the core topology intact, cover it with a wrap around configuration of sphere like sheets. This will provide greater conformity to the geometry of the spherical boundary. Write the appropriate TIL code, run the case, and examine the results at 100 and 200 sweeps respectively. Compare the results with those of problem 3, above.

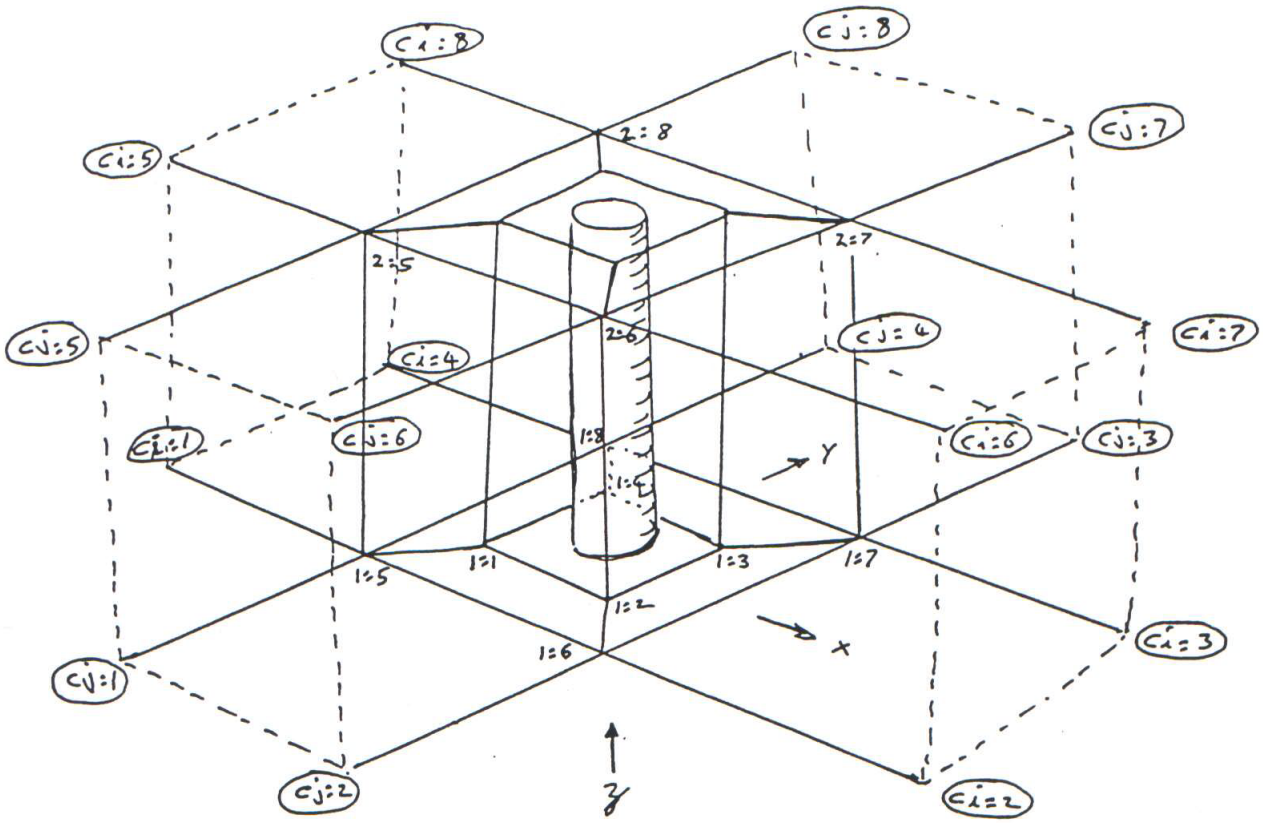
B.6 An Array of Rods





Program B.7 *Sub-program 2*

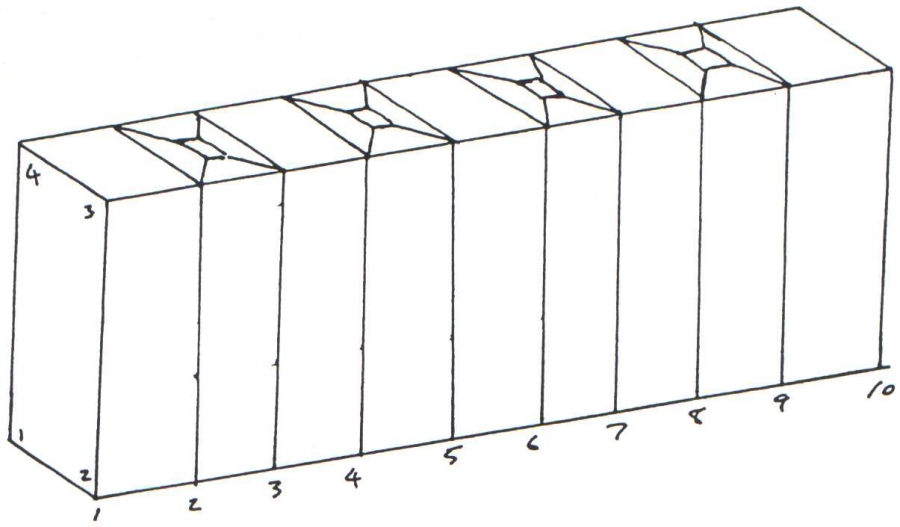
```
COMPONENT   hyperquad(sIN  s[1..4],   cIN  p[1..8],  c[1..8])
BEGIN
   c  1  -0.8   -0.8   0   -s  s:1 s:2  -L  p:1 ;
   c  2   0.8   -0.8   0   -s  s:1 s:2  -L  p:2 1;
   c  3   0.8    0.8   0   -s  s:1 s:2  -L  p:3 2;
   c  4  -0.8    0.8   0   -s  s:1 s:2  -L  p:4 3 1;

   c  5    -1     -1   0   -s  s:3 s:4  -L  p:5  c:1 c:5 1;
   c  6     1     -1   0   -s  s:3 s:4  -L  p:6  c:2 c:6 2 5;
   c  7     1      1   0   -s  s:3 s:4  -L  p:7  c:3 c:7 3 6;
   c  8    -1      1   0   -s  s:3 s:4  -L  p:8  c:4 c:8 4 7 5;
   x  f  1 3   5  7  ;
END
```

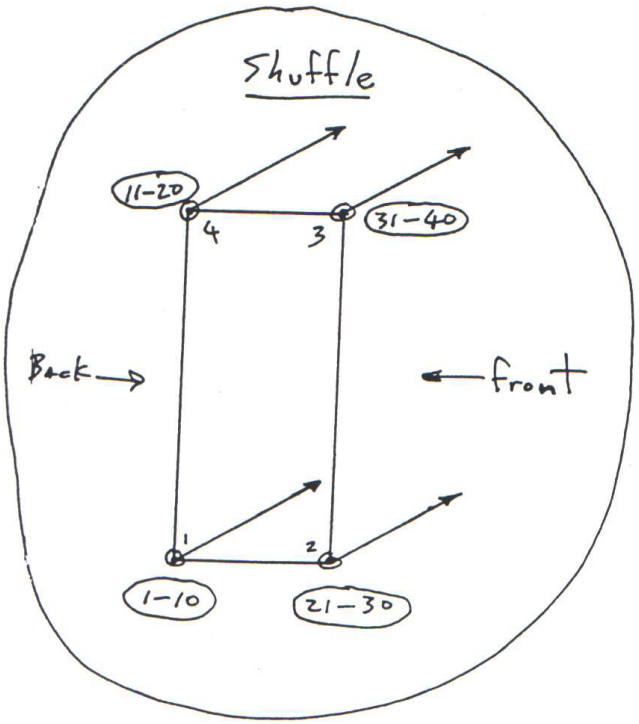


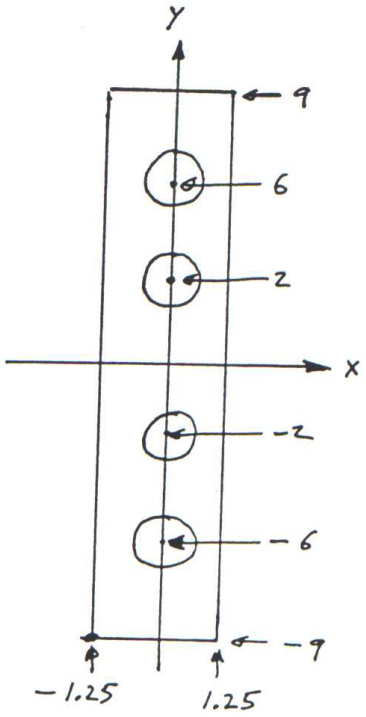
Program B.8 *Sub-program 3*

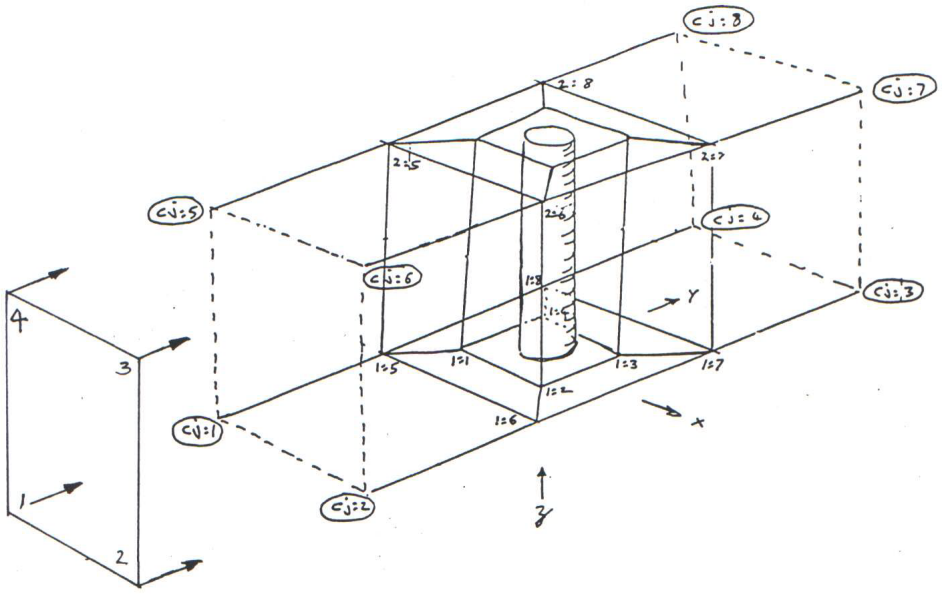
```
COMPONENT   Rod{sIN  sb, st,  ciN ci[1..8],  cj[1..8])
BEGIN
   s  1  -ellip(1  1  0);  #unit rod along the z-axis
```

```
    INPUT  1  (1.25 0 0   0 1.25 0   0 0 1) *
         hyperquad(sIN  (sb  1  sb  -1),  cIN(-8),
                        (ci:1..4   cj:1..4),  cOUT(1..8)) ;
    INPUT  2  (1.25 0 0   0 1.25 0   0 0 1) * (0 0 1)
         hyperquad(sIN  (st  1  st  -1),  cIN(1:1.,8),
                        (ci:5..8   cj:5..8),  cOUT(1..8)) ;
END
```









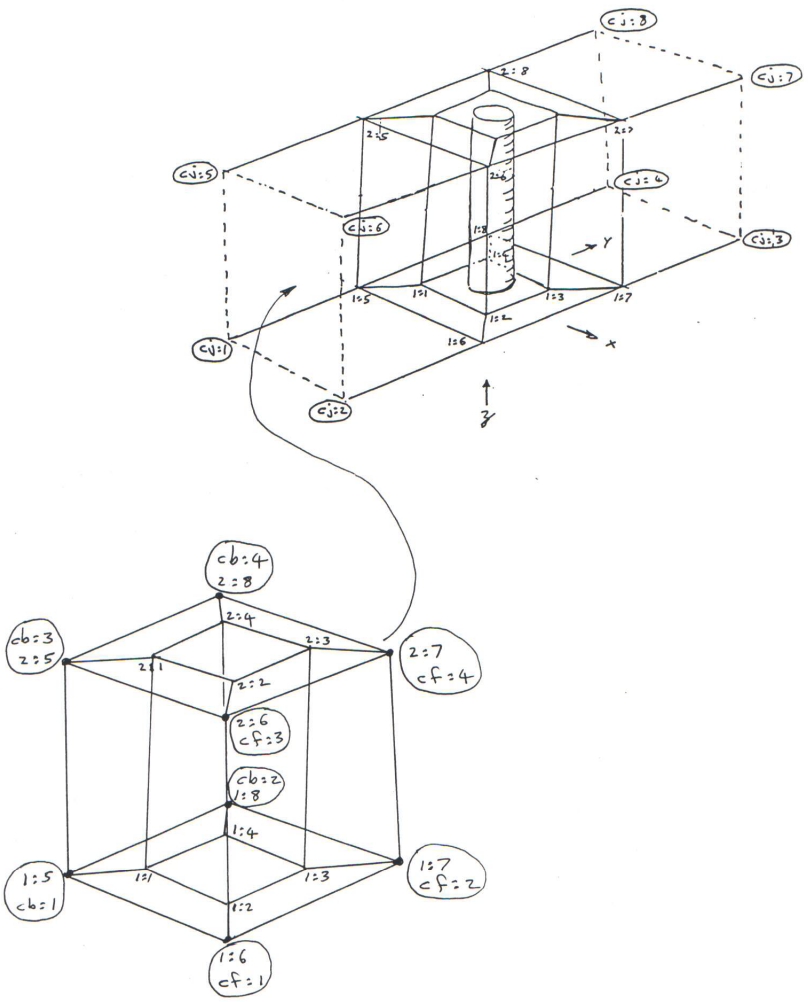
Program B.9 *Sub-program 4*

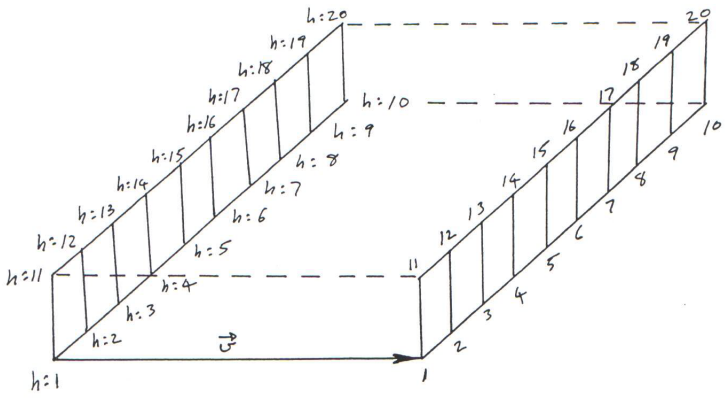
```
COMPONENT   RodRow4(sIN  sb, st, y1, y2,  cIN h[1..40])
BEGIN
   c  1  -1.25   -9   0   -s  sb y1  -L ;
   c  2   1.25   -9   0   -s  sb y1  -L  1 ;
   c  3   1.25   -9   1   -s  st y1  -L  2 ;
   c  4  -1.25   -9   1   -s  st y1  -L  3 1 ;

   INPUT 1 (0  -6  0) * Rod(sIN  (sb), (st),
            cIN (-8),  (l  2  -2  4  3  -2),
            cOUT cf(1:6..7  2:6..7), cb(l:5 1:8 2:5 2:8));
   INPUT 2 (0  -2  0) * Rod(sIN  (sb), (st),
            cIN (-8),  (lcb:2 lcf:2  -2  lcb:4 lcf:4  -2),
            cOUT cf(1:6..7  2:6..7), cb(l:5 1:8 2:5 2:8));
   INPUT 3 (0  2  0) * Rod(sIN  (sb), (st),
            cIN (-8),  (2cb:2 2cf:2  -2  2cb:4 2cf:4  -2),
            cOUT cf(1:6..7  2:6..7), cb(l:5 1:8 2:5 2:8));
   INPUT 4 (0  -6  0) * Rod(sIN  (sb), (st),
            cIN (-8),  (3cb:2 3cf:2  -2  3cb:4 3cf:4  -2),
            cOUT cf(1:6..7  2:6..7), cb(l:5 1:8 2:5 2:8));

   c  5  -1.25   9   0   -s  sb y2  -L 4cb:2 ;
   c  6   1.25   9   0   -s  sb y2  -L 4cb:2 5 ;
   c  7   1.25   9   1   -s  st y2  -L 4cb:4 6 ;
   c  8  -1.25   9   1   -s  st y2  -L 4cb:4 7 5 ;

   INPUT 5 shuffle(cIN
   (1  lcb:l..2  2cb:1..2  3cb:l..2  4cb:1..2  5
    4  1cb:3..4  2cb:3..4  3cb:3..4  4cb:3..4  8
```

```
        2  1cf:1..2  2cf:1..2  3cf:1..2  4cf:1..2  6
        3  1cf:3..4  2cf:3..4  3cf:3..4  4cf:3..4  7),
                     cOUT  (cs:1..40)) ;

        a    e
             5:1  h:21   5:2  h:22   5:3  h:23   5:4  h:24   5:5   h:25
             5:6  h:26   5:7  h:27   5:8  h:28   5:9  h:29   5:10  h:30
             5:11 h:31   5:12 h:32   5:13 h:33   5:14 h:34   5:15  h:35
             5:16 h:36   5:17 h:37   5:18 h:38   5:19 h:39   5:20  h:40
END
```

Program B.10 *Sub-program 5*

```
COMPONENT   shuffle(cIN  cs[1..40]))
BEGIN
END
```





Program B.11 *Sub-program 6*

```
COMPONENT   extrude(sIN  sb, st, x, y1, y2,  cIN v, h[1..20])
BEGIN
   c  1  @ <v> + <h:1>   -s  sb x y1  -L h:1   ;
   c  2  @ <v> + <h:2>   -s  sb x     -L h:2  1;
   c  3  @ <v> + <h:3>   -s  sb x     -L h:3  2;
   c  4  @ <v> + <h:4>   -s  sb x     -L h:4  3;
   c  5  @ <v> + <h:5>   -s  sb x     -L h:5  4;
   c  6  @ <v> + <h:6>   -s  sb x     -L h:6  5;
   c  7  @ <v> + <h:7>   -s  sb x     -L h:7  6;
   c  8  @ <v> + <h:8>   -s  sb x     -L h:8  7;
   c  9  @ <v> + <h:9>   -s  sb x     -L h:9  8;
   c  10 @ <v> + <h:10>  -s  sb x y2  -L h:10 9;

   c  11 @ <v> + <h:11>  -s  st x y1  -L h:11 1;
   c  12 @ <v> + <h:12>  -s  st x     -L h:12 2  11;
   c  13 @ <v> + <h:13>  -s  st x     -L h:13 3  12;
   c  14 @ <v> + <h:14>  -s  st x     -L h:14 4  13;
   c  15 @ <v> + <h:15>  -s  st x     -L h:15 5  14;
   c  16 @ <v> + <h:16>  -s  st x     -L h:16 6  15;
   c  17 @ <v> + <h:17>  -s  st x     -L h:17 7  16;
   c  18 @ <v> + <h:18>  -s  st x     -L h:18 8  17;
   c  19 @ <v> + <h:19>  -s  st x     -L h:19 9  18;
   c  20 @ <v> + <h:20>  -s  st x y2  -L h:20 10 19;
END
```

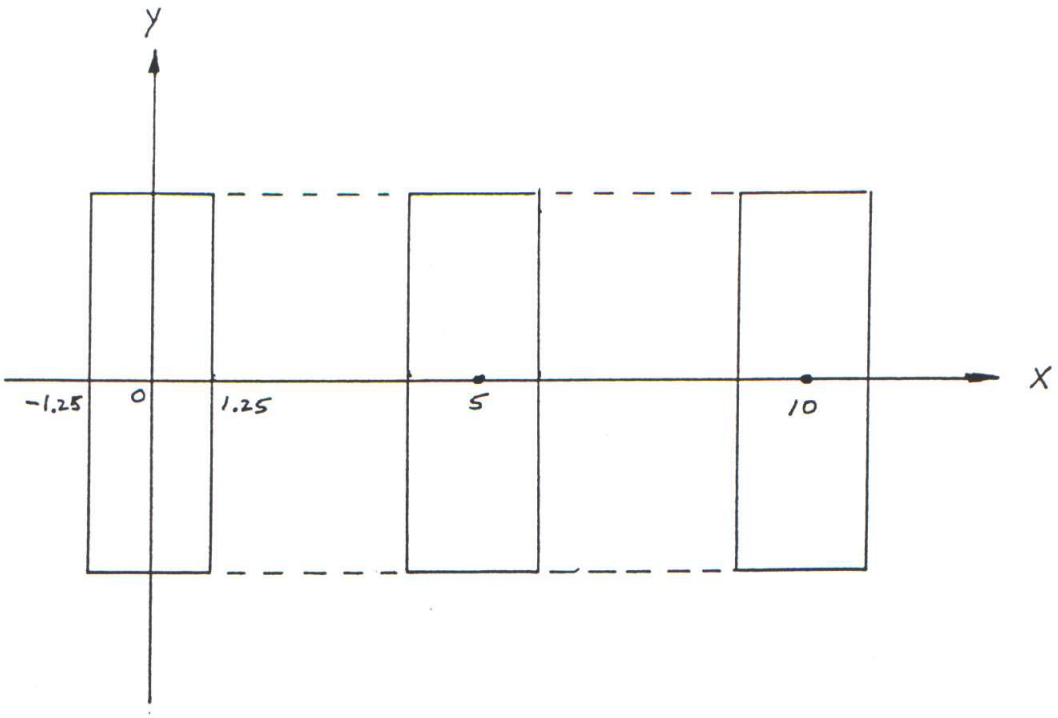
Program B.12 *Sub-program 7*

```
COMPONENT extrude_line(sIN  x, y1, y2, z,  cIN v, p[1..10], c[1..10])
BEGIN
   c  1  @ <v> + <c:1>   -s  x y1 z  -L p:1  c:1   ;
   c  2  @ <v> + <c:2>   -s  x    z  -L p:2  c:2  1;
   c  3  @ <v> + <c:3>   -s  x    z  -L p:3  c:3  2;
   c  4  @ <v> + <c:4>   -s  x    z  -L p:4  c:4  3;
   c  5  @ <v> + <c:5>   -s  x    z  -L p:5  c:5  4;
```

```
   c  6  @ <v> + <c:6>   -s  x    z  -L p:6  c:6  5;
   c  7  @ <v> + <c:7>   -s  x    z  -L p:7  c:7  6;
   c  8  @ <v> + <c:8>   -s  x    z  -L p:8  c:8  7;
   c  9  @ <v> + <c:9>   -s  x    z  -L p:9  c:9  8;
   c  10 @ <v> + <c:10>  -s  x y2 z  -L p:10 c:10 9;
END
```

Program B.13 *Sub-program 8*

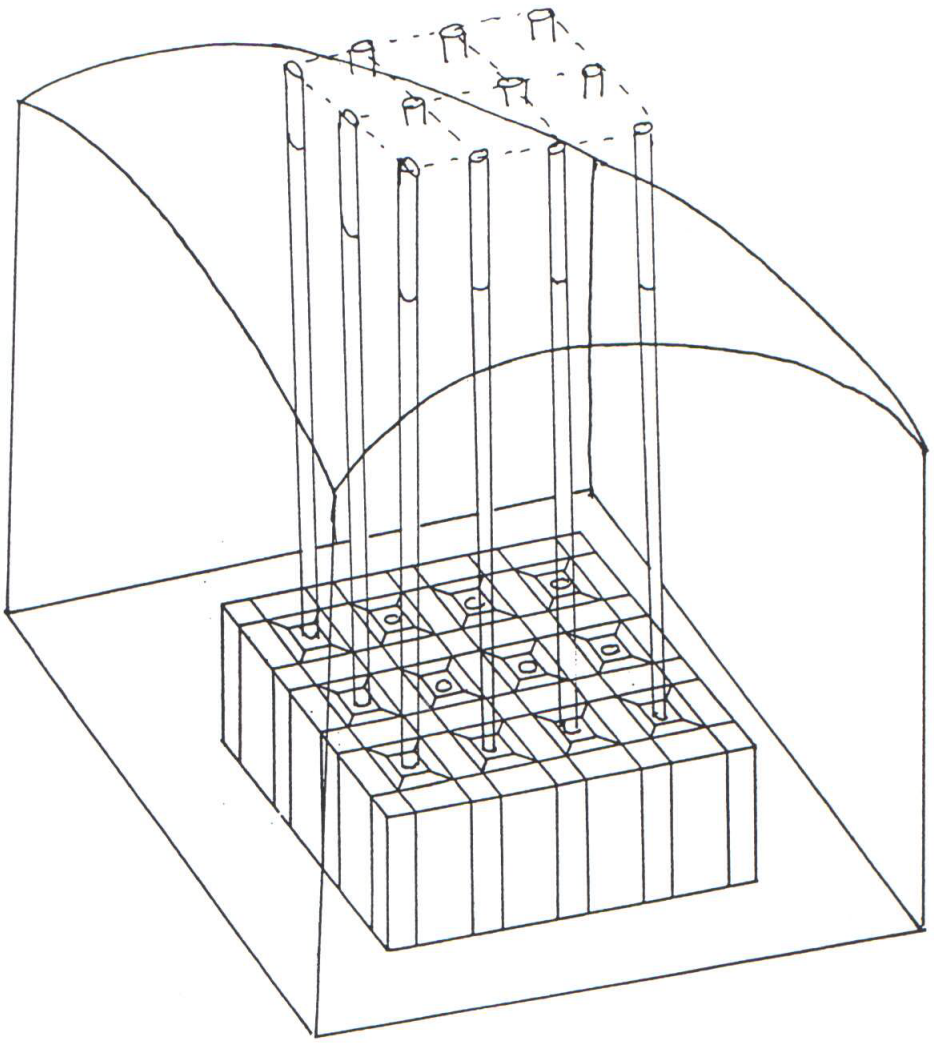
```
COMPONENT   extrude(sIN  sb, st, x, y1, y2,  cIN v, h[1..20])
BEGIN
   INPUT 1  extrude_line(sIN (x), (yl), (y2), (sb),
                         cIN (-10), (h;1..10), cOUT(1..10));
   INPUT 2  extrude_line(sIN (x), (yl), (y2), (st),
                         cIN (1:1..10), (h:11..20), cOUT(1..10));
END
```



Program B.14 *Sub-program 9*

```
COMPONENT   RodArray4x3(sIN  sb, st, x1, x2, y1, y2)
BEGIN
   VECTOR v;
   INPUT 1  RodRow4(sIN (sb), (st), (yl), (y2), cIN (-40),
                                                cOUT (1..40));
   INPUT 2  (5  0  0) *
            RodRow4(sIN (sb), (st), (yl), (y2), cIN (1:1..40),
```

```
                                                    cOUT (1..40));
   INPUT 3  (10 0  0) *
            RodRow4(sIN (sb), (st), (yl), (y2), cIN (2:1..40),
                                                    cOUT (1..40));
   <v>  = {-1, 0, 0);    #backward shift vector for sheet
   INPUT 4  extrude(sIN (sb), (st), (xl), (yl), (y2), cIN (v),
                                                       (1:1..20);
   <v>  = { 1, 0, 0);    #forward shift vector for sheet
   INPUT 5  extrude(sIN (sb), (st), (x2), (yl), (y2), cIN (v),
                                                      (3:21..40));
END
```



Program B.15 *Sub-program 10*

```
COMPONENT   RodsInBox()
BEGIN
   s  1  -plane(0    0    1   0) ;          #bottom boundary---sb
```

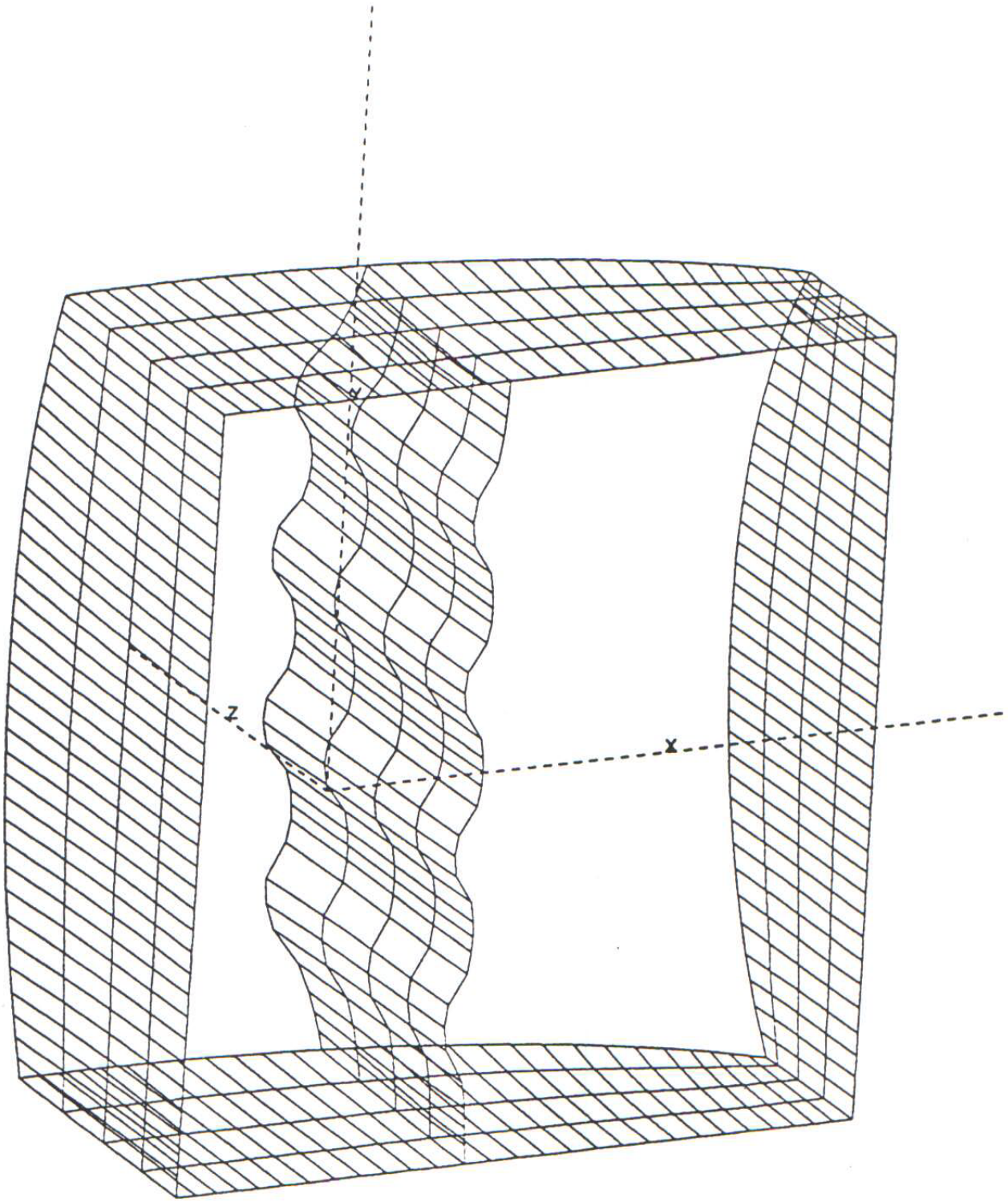
```
   s  2  -ellip(0.05 0.067 0.67) -t -1 0 0;   #top boundary------st
   s  3  -plane(1    0    3 ) ;               #left boundary-----x1
   s  4  -plane(-1   0    0   13) ;           #right boundary----x2
   s  5  -plane(0    1    0  9.5) ;           #near boundary-----y1
   s  4  -plane(0   -1    0  9.5) ;           #far boundary------y2
   INPUT  1 RodArray4x3(SIN (1), (2), (3), (4), (5), (6), cIN(-40));
END
```

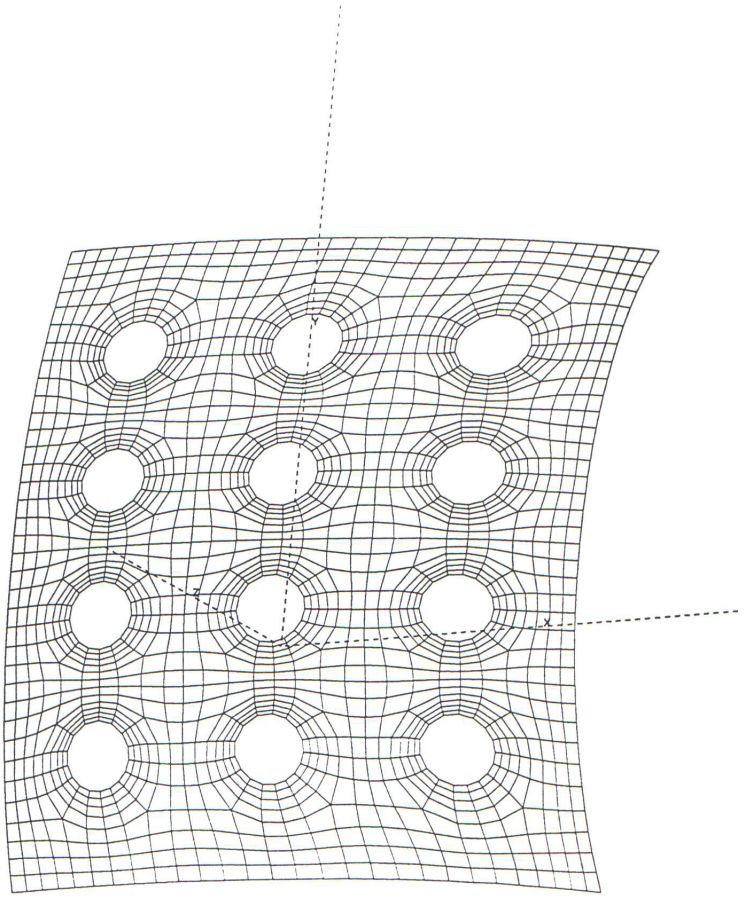
Program B.16 *File: 'rodarray.fra'*

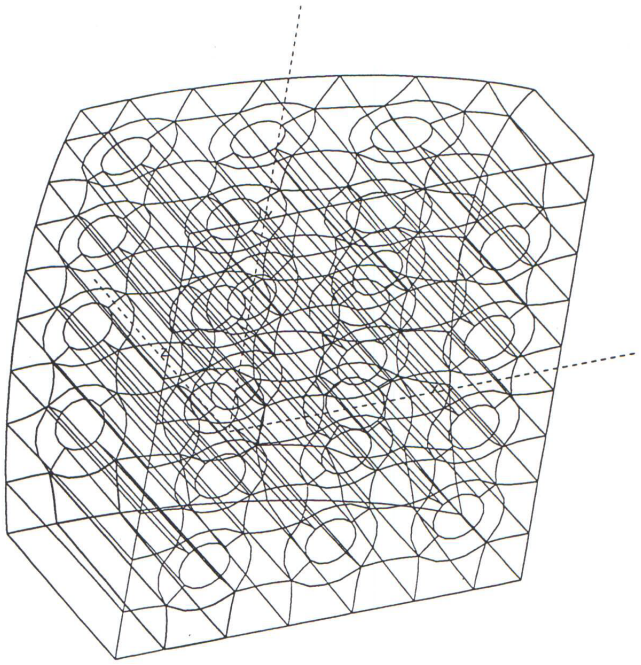
```
SET   GRIDDEN    4
COMPONENT   RodsInBox()
BEGIN
   s  1  -plane(0    0    1   0) ;            #bottom boundary---sb
   s  2  -ellip(0.05 0.067 0.67) -t -1 0 0;   #top boundary------st
   s  3  -plane(1    0    3 ) ;               #left boundary-----x1
   s  4  -plane(-1   0    0   13) ;           #right boundary----x2
   s  5  -plane(0    1    0  9.5) ;           #near boundary-----y1
   s  4  -plane(0   -1    0  9.5) ;           #far boundary------y2
   INPUT  1 RodArray4x3(SIN (1), (2), (3), (4), (5), (6), cIN(-40));
END

COMPONENT   RodArray4x3(sIN  sb, st, x1, x2, y1, y2)
BEGIN
   VECTOR v;
   INPUT 1  RodRow4(sIN (sb), (st), (yl), (y2), cIN (-40),
                                          cOUT (1..40));
   INPUT 2  (5  0  0) *
            RodRow4(sIN (sb), (st), (yl), (y2), cIN (1:1..40),
                                          cOUT (1..40));
   INPUT 3  (10 0  0) *
            RodRow4(sIN (sb), (st), (yl), (y2), cIN (2:1..40),
                                          cOUT (1..40));
   <v>  = {-1, 0, 0);    #backward shift vector for sheet
   INPUT 4  extrude(sIN (sb), (st), (xl), (yl), (y2), cIN (v),
                                              (1:1..20);
   <v>  = { 1, 0, 0);    #forward shift vector for sheet
   INPUT 5  extrude(sIN (sb), (st), (x2), (yl), (y2), cIN (v),
                                                (3:21..40));
END

COMPONENT   extrude(sIN  sb, st, x, y1, y2,  cIN v, h[1..20])
BEGIN
   INPUT 1  extrude_line(sIN (x), (yl), (y2), (sb),
                         cIN (-10), (h;1..10), cOUT(1..10));
   INPUT 2  extrude_line(sIN (x), (yl), (y2), (st),
                         cIN (1:1..10), (h:11..20), cOUT(1..10));
END
```

```
COMPONENT extrude_line(sIN  x, y1, y2, z,  cIN v, p[1..10], c[1..10])
BEGIN
   c  1  @ <v> + <c:1>   -s  x y1 z  -L p:1  c:1   ;
   c  2  @ <v> + <c:2>   -s  x    z  -L p:2  c:2  1;
   c  3  @ <v> + <c:3>   -s  x    z  -L p:3  c:3  2;
   c  4  @ <v> + <c:4>   -s  x    z  -L p:4  c:4  3;
   c  5  @ <v> + <c:5>   -s  x    z  -L p:5  c:5  4;
   c  6  @ <v> + <c:6>   -s  x    z  -L p:6  c:6  5;
   c  7  @ <v> + <c:7>   -s  x    z  -L p:7  c:7  6;
   c  8  @ <v> + <c:8>   -s  x    z  -L p:8  c:8  7;
   c  9  @ <v> + <c:9>   -s  x    z  -L p:9  c:9  8;
   c  10 @ <v> + <c:10>  -s  x y2 z  -L p:10 c:10 9;
END

COMPONENT   RodRow4(sIN  sb, st, y1, y2,  cIN h[1..40])
BEGIN
   c  1  -1.25   -9   0   -s  sb y1  -L ;
   c  2   1.25   -9   0   -s  sb y1  -L  1 ;
   c  3   1.25   -9   1   -s  st y1  -L  2 ;
   c  4  -1.25   -9   1   -s  st y1  -L  3 1 ;

   INPUT 1 (0  -6  0) * Rod(sIN  (sb), (st),
            cIN (-8),  (l  2  -2  4  3  -2),
            cOUT cf(1:6..7  2:6..7), cb(l:5 1:8 2:5 2:8));
   INPUT 2 (0  -2  0) * Rod(sIN  (sb), (st),
            cIN (-8),  (lcb:2 lcf:2  -2  lcb:4 lcf:4  -2),
            cOUT cf(1:6..7  2:6..7), cb(l:5 1:8 2:5 2:8));
   INPUT 3 (0  2  0) * Rod(sIN  (sb), (st),
            cIN (-8),  (2cb:2 2cf:2  -2  2cb:4 2cf:4  -2),
            cOUT cf(1:6..7  2:6..7), cb(l:5 1:8 2:5 2:8));
   INPUT 4 (0  -6  0) * Rod(sIN  (sb), (st),
            cIN (-8),  (3cb:2 3cf:2  -2  3cb:4 3cf:4  -2),
            cOUT cf(1:6..7  2:6..7), cb(l:5 1:8 2:5 2:8));

   c  5  -1.25   9   0   -s  sb y2  -L 4cb:2 ;
   c  6   1.25   9   0   -s  sb y2  -L 4cb:2 5 ;
   c  7   1.25   9   1   -s  st y2  -L 4cb:4 6 ;
   c  8  -1.25   9   1   -s  st y2  -L 4cb:4 7 5 ;

   INPUT 5 shuffle(cIN
   (1  lcb:l..2  2cb:1..2  3cb:l..2  4cb:1..2  5
    4  1cb:3..4  2cb:3..4  3cb:3..4  4cb:3..4  8
    2  1cf:1..2  2cf:1..2  3cf:1..2  4cf:1..2  6
    3  1cf:3..4  2cf:3..4  3cf:3..4  4cf:3..4  7),
                cOUT  (cs:1..40)) ;

    a  e
      5:1  h:21   5:2  h:22   5:3  h:23   5:4  h:24   5:5   h:25
```

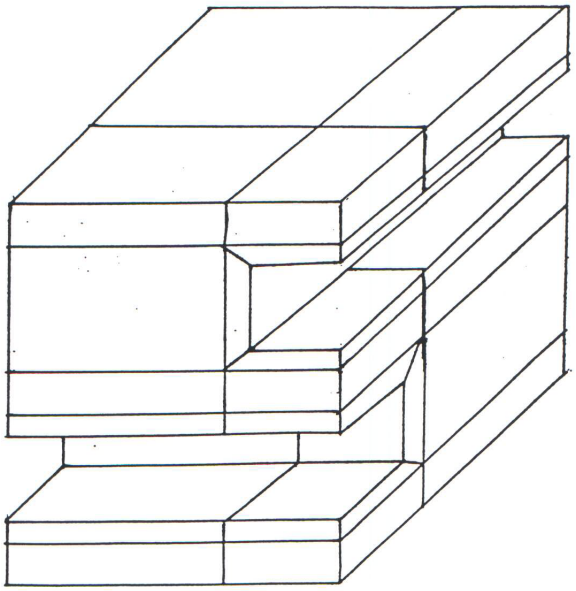
```
      5:6  h:26   5:7  h:27   5:8  h:28   5:9  h:29   5:10  h:30
      5:11 h:31   5:12 h:32   5:13 h:33   5:14 h:34   5:15  h:35
      5:16 h:36   5:17 h:37   5:18 h:38   5:19 h:39   5:20  h:40
END

COMPONENT   shuffle(cIN  cs[1..40]))
BEGIN
END

COMPONENT   Rod{sIN  sb, st,  ciN ci[1..8],  cj[1..8])
BEGIN
   s  1  -ellip(1  1  0);  #unit rod along the z-axis
   INPUT  1  (1.25 0 0   0 1.25 0   0 0 1) *
        hyperquad(sIN  (sb  1  sb  -1),  cIN(-8),
                       (ci:1..4   cj:l..4),  cOUT(1..8)) ;
   INPUT  2  (1.25 0 0   0 1.25 0   0 0 1) * (0 0 1)
        hyperquad(sIN  (st  1  st  -1),  cIN(1:1.,8),
                       (ci:5..8   cj:5..8),  cOUT(1..8)) ;
END

COMPONENT   hyperquad(sIN  s[1..4],   cIN  p[1..8],  c[1..8])
BEGIN
   c  1  -0.8   -0.8   0   -s  s:1 s:2  -L  p:1 ;
   c  2   0.8   -0.8   0   -s  s:1 s:2  -L  p:2 1;
   c  3   0.8    0.8   0   -s  s:1 s:2  -L  p:3 2;
   c  4  -0.8    0.8   0   -s  s:1 s:2  -L  p:4 3 1;

   c  5    -1     -1   0   -s  s:3 s:4  -L  p:5  c:1 c:5 1;
   c  6     1     -1   0   -s  s:3 s:4  -L  p:6  c:2 c:6 2 5;
   c  7     1      1   0   -s  s:3 s:4  -L  p:7  c:3 c:7 3 6;
   c  8    -1      1   0   -s  s:3 s:4  -L  p:8  c:4 c:8 4 7 5;
   x  f  1 3   5  7  ;
END
```

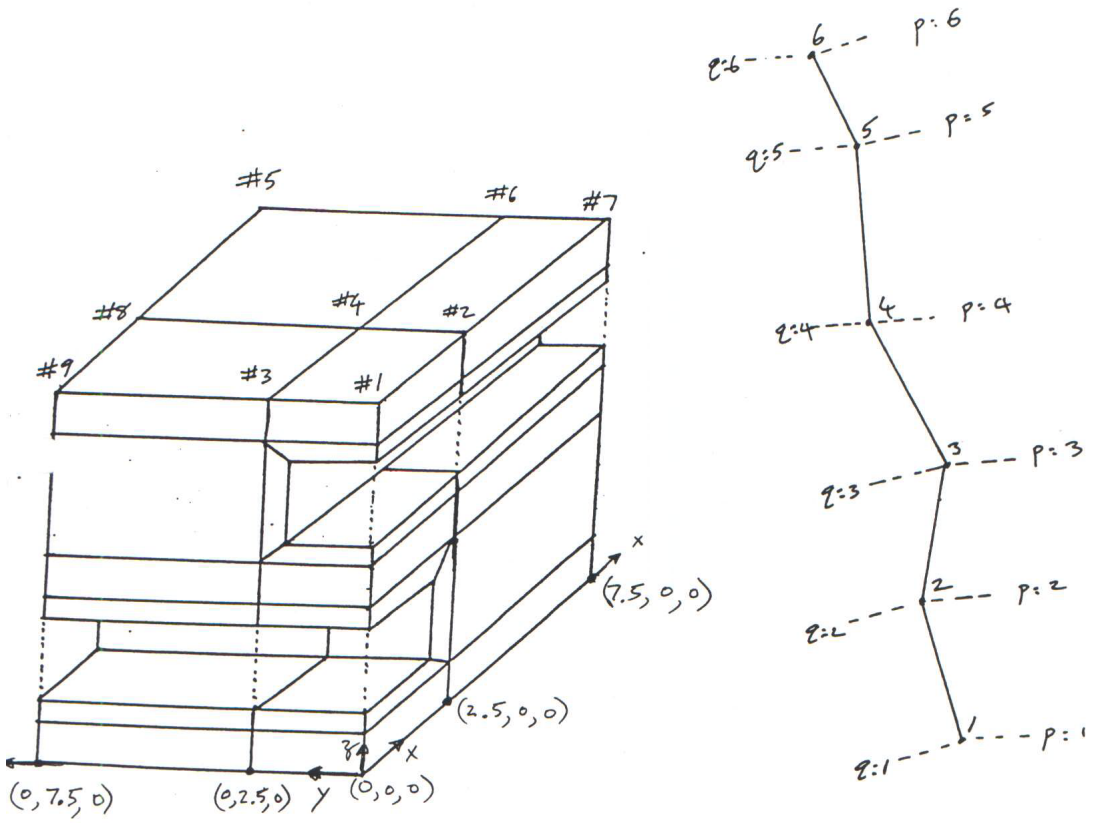

Z
X





B.7 Grid About Parallel Orthogonal Fibers





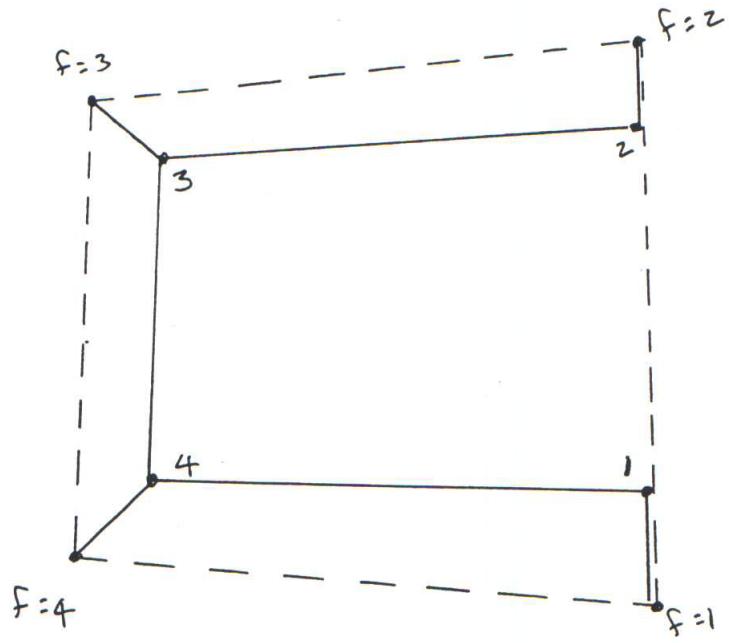
Program B.17 *Sub-program 11*

```
COMPONENT curve6(sIN  s[1..4], cIN t, x[1..6], p[1..6], q[1..6])
BEGIN
   c  1  @ <x:1> + <t>   -s  s:2 s:3 s:1  -L  p:1  q:1   ;
   c  2  @ <x:2> + <t>   -s  s:2 s:3      -L  p:2  q:2  1;
   c  3  @ <x:3> + <t>   -s  s:2 s:3      -L  p:3  q:3  2;
   c  4  @ <x:4> + <t>   -s  s:2 s:3      -L  p:4  q:4  3;
   c  5  @ <x:5> + <t>   -s  s:2 s:3      -L  p:5  q:5  4;
   c  6  @ <x:6> + <t>   -s  s:2 s:3 s:4  -L  p:6  q:6  5;
END
```

Program B.18 *Sub-program 12*

```
COMPONENT cubes_2x2x5(sIN  x0, x1, y0, y1, z0, z1)
BEGIN
   VECTOR x[1..6],  t
   <x:1> = (0, 0, 0  ) ;
   <x:2> = (0, 0, 2  ) ;
   <x:3> = (0, 0, 4.5) ;
   <x:4> = (0, 0, 5.5) ;
   <x:5> = (0, 0, 8  ) ;
   <x:6> = (0, 0, 10 ) ;

   #Rod Crossover Location
   INPUT 1  curve6(sIN  (z0 x0 y0 zl),
                   cIN  (-1),(x:1..6), (-6), (-6),
                   cOUT (1..6));

   <t>  =  {2.5, 0, 0} ;
   INPUT 2  curve6(sIN  (z0 -1 y0 zl),
                   cIN  (t), (x:1..6), (1:1..6), (-6),
                   cOUT (1..6));

   <t>  =  {0, 2.5, 0} ;
   INPUT 3  curve6(sIN  (z0 x0 -1 zl),
                   cIN  (t), (x:1..6), (1:1..6), (-6),
                   cOUT (1..6));

   <t>  =  {2.5, 2.5, 0};
   INPUT 4  curve6(sIN (z0 -1 -1 zl),
                   cIN  (t), (x:1..6), (2:1..6), (3:1..6),
                   cOUT (1..6));

   #Diagonally Opposite Corner from Crossover Location
   <t>  =  {7.5, 7.5, 0) ;
   INPUT 5  curve6(sIN (z0 xl yl zl),
                   cIN  (-1), (x:1..6), (-6), (-6),
                   cOUT (1..6));
```

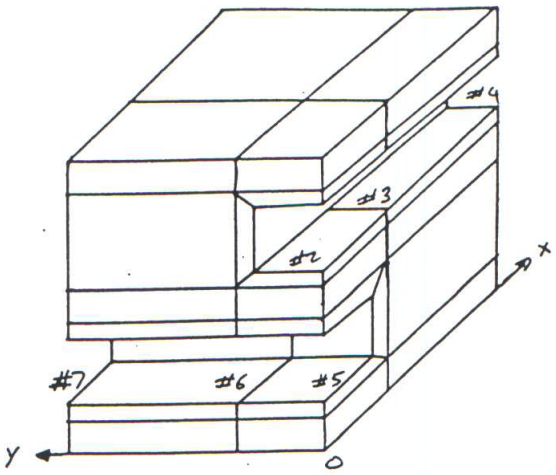
```
   #The xmax boundary b2
   <t>  =  {7.5, 2.5, 0} ;
   INPUT 6  curve6(sIN (z0 xl -1 zl),
                   cIN  (t), (x:1..6), (4:1..6), (5:1..6),
                   cOUT (1..6));

   <t>  =  {7.5, 0, 0} ;
   INPUT 7  curve6(sIN (z0 xl y0 zl),
                   cIN  (t), (x:1..6), (2:1..6), (6:1..6),
                   cOUT (1..6));

   #The ymax boundary b4
   <t>  =  {2.5, 7.5, 0} ;
   INPUT 8  curve6(sIN (z0 yl -1 zl),
                   cIN  (t), (x:1..6), (4:1..6), (5:1..6),
                   cOUT (1..6));
   <t>  =  {0, 7.5, 0} ;
   INPUT 9  curve6(sIN (z0 x0 yl zl),
                   cIN  (t), (x:1..6), (3:1..6), (8:1..6),
                   cOUT (1..6));
END
```



Program B.19 *Sub-program 13*

```
COMPONENT semi_loop_face(sIN  s[1..3], cIN f[1..4], p[1..4], q[1..6])
BEGIN
```

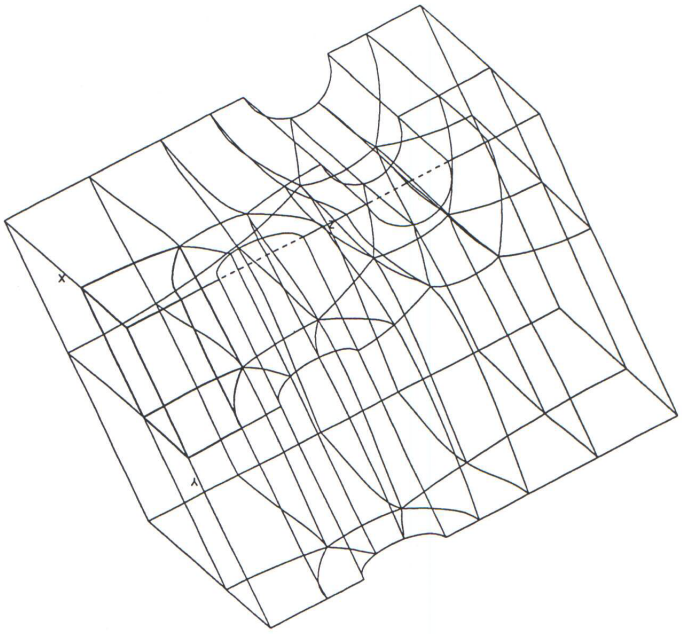
```
   c  1  @ 0.9*<f:1>+0.1*<f:2>   -s  s:1 s:2 s:3  -L  f:1 p:1  q:1   ;
   c  2  @ 0.1*<f:1>+0.9*<f:2>   -s  s:1 s:2 s:3  -L  f:2 p:2  q:2   ;
   c  3  @ 0.1*<f:1>+0.9*<f:3>   -s  s:1     s:3  -L  f:3 p:3  q:3  2;
   c  4  @ 0.1*<f:2>+0.9*<f:4>   -s  s:1     s:3  -L  f:4 p:4  q:4  3 1;
   x  e  f:1   f:2  ;
END
```

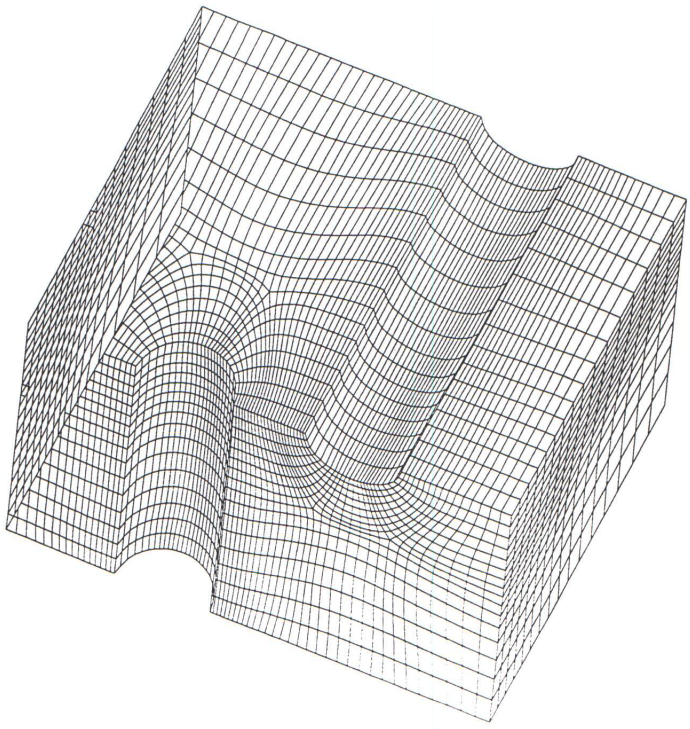


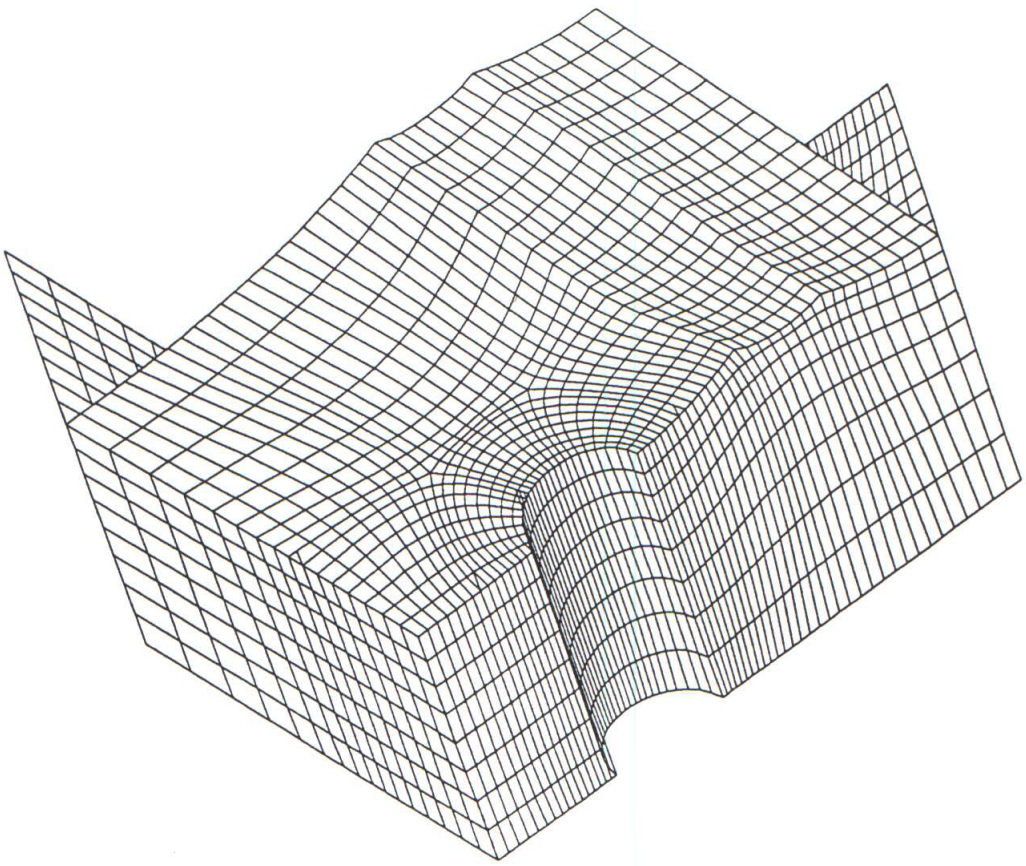
Program B.20 *Sub-program 14*

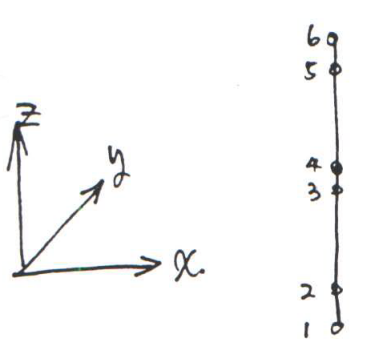
```
COMPONENT   fibre()
BEGIN
   s  1 -plane @ ({1, 0, 0}, {0, 0, 0})  ;     # xmin boundary x0
   s  2 -plane @ ({-1, 0, 0}, {8, 0, 0}) ;     # xmax boundary xl
   s  3 -plane @ ({0, 1, 0}, {0, 0, 0})  ;     # ymin boundary y0
   s  4 -plane @ ({0, 1, 0}, {0, 8, 0})  ;     # ymax boundary yl
   s  5 -plane @ ({0, 0, 1}, {0, 0, 0})  ;     # zmin boundary z0
   s  6 -plane @ ({0, 0, -1}, {0,0, 10}) ;     # zmax boundary zl
   s  7 -ellip (0, 1, 1) -t @ ({0, 0, 6.75));  # fiber in x direction
   s  8 -ellip (1, 0, 1) -t @ ({0, 0, 3.25));  # fiber in y direction
   INPUT 1  cubes_2x2x5(sIN (1), (2), (3), (4), (5), (6),
                                      cOUT  xl(1:4 1:5 3:5 3:4)
                                            x2(2:4 2:5 4:5 4:4)
                                            x3(7:4 7:5 6:5 6:4)
                                            yl(1:5 1:3 2:3 2:2)
                                            y2(3:2 3:3 4:3 4:2)
                                            y3(9:2 9:3 8:3 8:2));
   INPUT 2  semi_loop_face(sIN( 1 3 7), cIN(1x1:1..4), (-4),     (-4),
                                          cOUT (1..4));
   INPUT 3  semi_loop_face(sIN(-1 3 7), cIN(1x2:1..4), (2:1..4), (-4),
                                          cOUT (1..4));
   INPUT 4  semi_loop_face(sIN(2 3 7), cIN(1x3:1..4), (3:1..4),  (-4),
                                          cOUT (1..4));
   INPUT 5  semi_loop_face(sIN(3 1 8), cIN(lyl:1..4), (-4),      (-4),
```

```
                                                cOUT (1..4));
   INPUT 6  semi_loop_face(sIN(-1 1 8), cIN(ly2:1..4), (5:1..4), (-4),
                                                cOUT (1..4));
   INPUT 7  semi_loop_face(sIN(4 1 8), cIN(ly3:1..4), (6:1,.4), (-4));
END
```

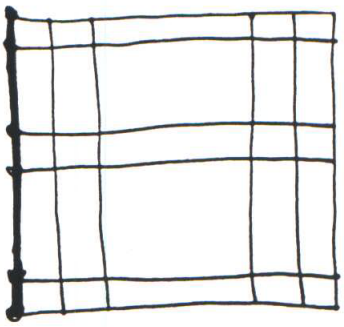




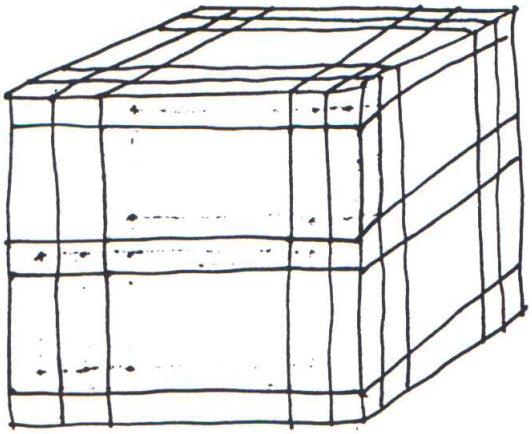




curve6( )

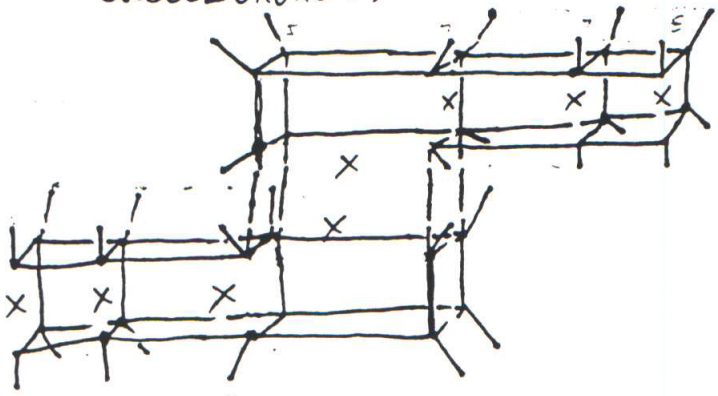


sheet_6x6( )



cubes_6x6x6( )

semi_loop( )



oneFiber( )

B.8 Fiber

Program B.21 *File: 'fiber1.fra'*

```
COMPONENT   fiber()
BEGIN
   VECTOR d;
   s  1 -plane @ ({1, 0, 0}, {0, 0, 0})  ;     # xmin boundary x0
   s  2 -plane @ ({-1, 0, 0}, {8, 0, 0}) ;     # xmax boundary xl
   s  3 -plane @ ({0, 1, 0}, {0, 0, 0})  ;     # ymin boundary y0
   s  4 -plane @ ({0, 1, 0}, {0, 8, 0})  ;     # ymax boundary yl
   s  5 -plane @ ({0, 0, 1}, {0, 0, 0})  ;     # zmin boundary z0
   s  6 -plane @ ({0, 0, -1}, {0,0, 10}) ;     # zmax boundary zl
   s  7  -tube "fy0.dat" ;  # fiber on y0 face
   s  8  -tube "fx0.dat" ;  # fiber on x0 face
   s  9  -tube "fy1.dat" ;  # fiber on y1 face
   s  10 -tube "fx1.dat" ;  # fiber on x1 face

   INPUT 1  cubes_6x6x6(sIN (1), (2), (3), (4), (5), (6),
            cOUT fy0(1:3  1:2  2:2  2:3    1:9  1:8  2:8  2:9
                     1:15 1:14 2:14 2:15   1:21 1:20 2:20 2:21
                     1:17 1:16 2:16 2:17   1:23 1:22 2:22 2:23
                     1:29 1:28 2:28 2:29   1:35 1:34 2:34 2:35),
                 fx0(6:3 6:2 6:8  6:9   5:3 5:2 5:8  5:9
                     4:3 4:2 4:8  4:9   3:3 3:2 3:8  3:9
                     4:5 4:4 4:10 4:11  3:5 3:4 3:10 3:11
                     2:5 2:4 2:10 2:11  1:5 1:4 1:10 1:11),
                 fy1(6:33 6:32 5:32 5:33   6:27 6:26 5:26 5:27
                     6:21 6:20 5:20 5:21   6:15 6:14 5:14 5:15
                     6:23 6:22 5:22 5:23   6:17 6:16 5:16 5:17
                     6:11 6:10 5:10 5:11   6:5  6:4  5:4  5:5);
                 fx1(1:33 1:32 1:26 1:27   2:33 2:32 2:26 2:27
                     3:33 3:32 3:26 3:27   4:33 4:32 4:26 4:27
                     3:35 3:34 3:28 3:29   4:35 4:34 4:28 4:29
                     5:35 5:34 5:28 5:29   6:35 6:34 6:28 6:29));
   <d>  =  {0.1,  0,  0} ;
   INPUT 2  oneFiber(sIN (7), (3), (1..2),
     cIN (d), (1fy0:1..4),   (1fy0:5..8),   (1fy0:9..12),  (1fy0:13..16),
              (1fy0:17..20), (1fy0:21..24), (1fy0:25..28), (1fy0:29..32));
   <d>  =  {0, -0.1,  0} ;
   INPUT 3  oneFiber(sIN (8), (1), (4  3),
     cIN (d), (1fx0:1..4),   (1fx0:5..8),   (1fx0:9..12),  (1fx0:13..16),
              (1fx0:17..20), (1fx0:21..24), (1fx0:25..28), (1fx0:29..32));
   <d>  =  {-0.1,  0,  0} ;
   INPUT 4  oneFiber(sIN (9), (4), (2  1),
     cIN (d), (1fy1:1..4),   (1fy1:5..8),   (1fy1:9..12),  (1fy1:13..16),
              (1fy1:17..20), (1fy1:21..24), (1fy1:25..28), (1fy1:29..32));
   <d>  =  {0, 0.1,  0} ;
   INPUT 5  oneFiber(sIN (10), (2), (3  4),
```

```
             cIN (d), (1fx1:1..4),   (1fx1:5..8),   (1fx1:9..12),  (1fx1:13..16),
                      (1fx1:17..20), (1fx1:21..24), (1fx1:25..28), (1fx1:29..32));
          END
#-------------------------------------------------------------------------------#
# Sub-COMPONENTs (must be after the main COMPONENT)

          COMPONENT   oneFiber(sIN  fib, wall, end[1..2],
                      cIN d, sec1[1..4], sec2[1..4], sec3[1..4], sec4[1..4],
                             sec5[1..4], sec6[1..4], sec7[1..4], sec8[1..4])
          BEGIN
             VECTOR D, zero;
             <D>  =  -<d>; <zero>  =  [0, 0, 0] ;
             x   e   sec1:1  sec1:2  sec2:1  sec2:2  sec3:1  sec3:2  sec4:1  sec3:1;
             x   e   sec5:2  sec6:2  sec6:1  sec6:2  sec7:1  sec7:2  sec8:1  sec8:2;
             INPUT 1  semi_loop(sIN (fib), (wall), (end:1), cIN (zero), (sec1:1..4),
                                                                 (-4), cOUT (1..4));
             INPUT 2  semi_loop(sIN (fib), (wall), (-1),    cIN (zero), (sec2:1..4),
                                                             (1:1..4), cOUT (1..4));
             INPUT 3  semi_loop(sIN (fib), (wall), (-1),    cIN (d),    (sec3:1..4),
                                                             (2:1..4), cOUT (1..4));
             INPUT 4  semi_loop(sIN (fib), (wall), (-1),    cIN (D),    (sec4:1..4),
                                                             (3:1..4), cOUT (1..4));
             a   e   3:1  5:2  4:1  6:2  3:4  5:3  4:4  6:3;
             x   e   5:2  6:2;
             a   e   5:1  5:2;
          END

          COMPONENT cubes_6x6x6(sIN  x0, x1, y0, y1, z0, z1)
          BEGIN
             VECTOR y[1..6],
             <y:1>  =  {0,  0,  0}; <y:2>  =  {0,  0.5,  0}, <y:3>  =  {0,  1,  0};
             <y:4>  =  {0,  7,  0}; <y:5>  =  {0,  7.5,  0}, <y:6>  =  {0,  8,  0};

             INPUT 1  sheet_6x6(sIN (x0), (x1), (y0), (z0), (z1), cIN (y:1), (-36),
                      cOUT (1:1..6 2:1..6 3:1..6 4:1..6 5:1..6 6:1..6));
             INPUT 2  sheet_6x6(sIN (x0), (x1), (-1), (z0), (z1), cIN (y:2), (1:1..36),
                      cOUT (1:1..6 2:1..6 3:1..6 4:1..6 5:1..6 6:1..6));
             INPUT 3  sheet_6x6(sIN (x0), (x1), (-1), (z0), (z1), cIN (y:3), (2:1..36),
                      cOUT (1:1..6 2:1..6 3:1..6 4:1..6 5:1..6 6:1..6));
             INPUT 4  sheet_6x6(sIN (x0), (x1), (-1), (z0), (z1), cIN (y:4), (3:1..36),
                      cOUT (1:1..6 2:1..6 3:1..6 4:1..6 5:1..6 6:1..6));
             INPUT 5  sheet_6x6(sIN (x0), (x1), (-1), (z0), (z1), cIN (y:5), (4:1..36),
                      cOUT (1:1..6 2:1..6 3:1..6 4:1..6 5:1..6 6:1..6));
             INPUT 3  sheet_6x6(sIN (x0), (x1), (y1), (z0), (z1), cIN (y:6), (5:1..36),
                      cOUT (1:1..6 2:1..6 3:1..6 4:1..6 5:1..6 6:1..6));
          END

          COMPONENT sheet_6x6(sIN  x0, x1, sy, z0, z1, cIN y, sh[1..36])
```

```
BEGIN
   VECTOR z[1..6], x[1..6];
   <x:1>  =  {0,  0,  0}; <x:2>  =  {0.5,  0,  0}, <x:3>  =  {1,  0,  0};
   <x:4>  =  {7,  0,  0}; <x:5>  =  {7.5,  0,  0}, <x:6>  =  {8,  0,  0};
   <z:1>  =  {0, 0, 0}   +  <y> ; <z:2>  =  {0, 0, 1}   +  <y> ;
   <z:3>  =  {0, 0, 4.5} +  <y> ; <z:4>  =  {0, 0, 5.5} +  <y> ;
   <z:5>  =  {0, 0, 9}   +  <y> ; <z:6>  =  {0, 0, 10}  +  <y> ;

   INPUT 1  curve6(sIN  (z0 x0 sy z1),
                   cIN  (x:1), (z:1..6), (-6), (sh:1..6),       cOUT(1..6));
   INPUT 2  curve6(sIN  (z0 -1 sy z1),
                   cIN  (x:2), (z:1..6), (1:1..6), (sh:7..12),  cOUT(1..6));
   INPUT 3  curve6(sIN  (z0 -1 sy z1),
                   cIN  (x:3), (z:1..6), (2:1..6), (sh:13..18), cOUT(1..6));
   INPUT 4  curve6(sIN  (z0 -1 sy z1),
                   cIN  (x:4), (z:1..6), (3:1..6), (sh:19..24), cOUT(1..6));
   INPUT 5  curve6(sIN  (z0 -1 sy z1),
                   cIN  (x:5), (z:1..6), (4:1..6), (sh:25..30), cOUT(1..6));
   INPUT 6  curve6(sIN  (z0 x1 sy z1),
                   cIN  (x:6), (z:1..6), (5:1..6), (sh:31..36), cOUT(1..6));
END

COMPONENT semi_loop_face(sIN  fib, wall, end, cIN d, f[1..4], p[1..4])
BEGIN
   c  1  @ 0.9*<f:1>+0.1*<f:2>   -s  fib end wall  -L  f:1 p:1  ;
   c  2  @ 0.1*<f:1>+0.9*<f:2>   -s  fib end wall  -L  f:2 p:2  ;
   c  3  @ 0.1*<f:1>+0.9*<f:3>   -s  fib end       -L  f:3 p:3  2;
   c  4  @ 0.1*<f:2>+0.9*<f:4>   -s  fib end       -L  f:4 p:4  3 1;
END

COMPONENT curve6(sIN  s[1..4], cIN t, x[1..6], p[1..6], q[1..6])
BEGIN
   c  1  @ <x:1> + <t>   -s  s:2 s:3 s:1  -L  p:1  q:1   ;
   c  2  @ <x:2> + <t>   -s  s:2 s:3      -L  p:2  q:2  1;
   c  3  @ <x:3> + <t>   -s  s:2 s:3      -L  p:3  q:3  2;
   c  4  @ <x:4> + <t>   -s  s:2 s:3      -L  p:4  q:4  3;
   c  5  @ <x:5> + <t>   -s  s:2 s:3      -L  p:5  q:5  4;
   c  6  @ <x:6> + <t>   -s  s:2 s:3 s:4  -L  p:6  q:6  5;
END
```

Program B.22 *File: 'fiber1.sch'*

```
step 100:  -w  50  -S  51
write -a  -f  blk.tmp
write -a  -D  0  -f  dump.tmp
```