FCONVERT

FCONVERT is primarily used as a pre-processing tool for DPLR. Its main function is to read a plot3d grid file as input and convert it into a format that can be used by the DPLR CFD codes. During the conversion process the file can be manipulated in various ways. For instance, grid files can be scaled by a multiplicative factor, “sequenced” or coarsened in one or more dimensions, or decomposed into multiple pieces in order to run on a parallel machine. FCONVERT is also used to process input radiation and boundary condition (BC) files for use by DPLR, and can be used to change the format of a restart file, or to convert a plot3d flow file into a restart file. All of these uses will be discussed in detail in this chapter. Unlike DPLR2D and DPLR3D, FCONVERT is a serial code; all pre-processing must be done on a single processor.

Running FCONVERT

FCONVERT is run from the command line. The user first prepares the input deck, either by starting from a similar case or by following the rules discussed in the following section. When a new simulation is begun, FCONVERT is used to process the grid file for use by DPLR. FCONVERT may also be used to process input radiation or boundary condition files, if any. In order to execute the run, type:

```
fconvert < fcon.inp
```

at the command line, where “`fcon.inp`” is the name of the input deck. When FCONVERT is executed diagnostic output will be echoed to the screen in order to provide feedback on the action(s) being performed. Any warning messages will also be echoed to the screen. If a fatal error is detected during execution, a descriptive message will be echoed to the screen and execution will terminate.

The sample input deck presented in the following section replicates that for one of the sample problems (“Neptune”) provided with this distribution. When FCONVERT is executed on this sample problem, the output is:

```
*********************************************
fconvert
NASA Ames Version 3.05.0
Mike Wright     last modified: 03/31/06
*********************************************


Reading plot3d   ascii         file  neptune.g
Writing archival XDR-formatted file  neptune.pgrx

Input  file does not include dummy cells
```

```
Output file includes dummy cells


Input file is 3D

Input Block 1 size: il = 32; jl = 16; kl = 64  ( 32768 cells)
Input Block 2 size: il = 48; jl = 64; kl = 64  (196608 cells)

Largest block is:
     nb =   2; original block =   2
     il =  48; jl =  64; kl =  64


Read input interface file neptune.inter
Found   3 valid zonal interface blocks in   2 block grid file


Decomposing block   1 into   1:   ibrk=  1 jbrk=  1 kbrk=  1
Decomposing block   2 into   3:   ibrk=  3 jbrk=  1 kbrk=  1
-----------------------------------------
                  creating   4 total blocks

  4 Blocks;  Max load imbalance =  12.50%

Output Block 1 size: il = 32; jl = 16; kl = 64  ( 32768 cells)
Output Block 2 size: il = 16; jl = 64; kl = 64  ( 65536 cells)
Output Block 3 size: il = 16; jl = 64; kl = 64  ( 65536 cells)
Output Block 4 size: il = 16; jl = 64; kl = 64  ( 65536 cells)

Largest block is:
     nb =   2; original block =   2
     il =  16; jl =  64; kl =  64

---------------------------------------------
Summary (grid dimensions for CFD input deck):
Hardwired to run on   4 processors

     Block   1; nx =  32; ny =  16; nz =  64
     Block   2; nx =  48; ny =  64; nz =  64

==> Finished writing output file neptune.pgrx

Done!
```

As can be seen, the screen output provides a step-by step discussion of the actions performed during execution, and can be used to ensure that the actions performed match those desired.

Additional data, such as final load imbalance and block size information, are also output. Note that block size and load imbalance information is provided in terms of the number of cells in each computational direction, NOT the number of points. Finally, if the output format is parallel, the code outputs master block sizes to be used as grid dimensions in the CFD code input deck.

Sample Input Deck

A sample input deck for FCONVERT is shown below. A brief description of each of the flags, along with allowable settings, is provided in this section. Detailed discussions of some of the more complex options follow. Additional examples of FCONVERT input decks can be found in the sample problems that are distributed with the DPLR package; it is recommended that the user run through each of the provided examples after reading this chapter and examine the output of FCONVERT for each case.

Some of the flags discussed below have options that are meant to be used for debugging purposes. These settings are intended for developers of the software, rather than users, and can be safely ignored by those who do not intend to modify the source code. A description of their function is provided primarily for completeness.

```
Input file for fconvert

iaction ifile   idim   iinfo   ivers   nvers
  1       1       3      0       1      3.05

inform  inint  idummy  nborig
 22       1       0       2

ouform  ouint  odummy  ncedge
 11       0       0       1

imseq   iscale   sfact
  0       0      1.0

nbreak
  1

Decomposition information for each master block
ibrk    jbrk    kbrk
  1       1       1
  3       1       1

Sequencing information for each master block
iseq    jseq    kseq
 -1       1       1
```

```
iname,xname
'neptune'
'neptune'

oname
'neptune'

nsin  nerin  nevin  necin  ntbin
 5     0       1      0      0
```

Summary of Input Flags

Input flags will be described in the order in which they appear in the input deck. A full description of some of the more complex option will be deferred until later sections.

`iaction`

Specifies the action to perform. Allowable values of `iaction` are:

0	test decompose over a range of blocks
1	decompose file according to `(ijk)brk`
2	decompose file according to `nbreak`
3	recompose file into original blocks
10	format conversion only
11	stop after printing file size

`iaction` should be set to 10 when no parallel decomposition or recomposition is desired. In this mode FCONVERT will simply read the input file, scale or sequence it if desired, and write the output in the chosen format. When `iaction` = 11 the code will read the input file, determine the dimensions and the number of computational cells in each block, and output this information to the screen. No output file will be generated. See below for more information on the other possible actions.

`ifile`

Specifies the type of file to be processed. Allowable values of `ifile` are:

1	grid file
2	restart/flow file
3	boundary condition (BC) file
4	radiation file

restart, BC, and radiation files are never decomposed or recomposed; therefore the only allowable actions for these file types are iaction = 10 (format conversion) and iaction = 11 (file size).

idim

Specifies the dimension of the input file. Allowable values of idim are:

2	2D/Axisymmetric
3	3D

Note that it is necessary to use input and output files that are of the proper dimension for the desired simulation. For example, if it is desired to run DPLR2D on an axisymmetric problem, the input grid file to FCONVERT must be two-dimensional. FCONVERT does not strip the third dimension from an input file, and will generate unusable output if it is given a file of different dimension than specified by the idim flag.

iinfo

Controls the output of debugging information. This information is intended for software developers. Allowable values of iinfo are:

0	Do not output debugging information
1	Output debugging information

ivers

Specify the output file version. All parallel grid, restart, BC, and radiation files are version specific. When the major or minor release number of the package changes, one or more of these files could have changes in their format. The release notes for the package will indicate whether a file format has changed in the latest release. While the entire package maintains backward compatibility and can always read and process files created with previous versions of the software, the reverse is of course not true. The ivers flag allows format conversion of all files between all versions. This allows the user to run DPLR using input files that were originally created with an older version of the package, for example. Allowable values of ivers are:

1	Do not attempt to change file version
2	Upgrade file to the current version

3	Convert file to the version specified by `nvers`

The version number is a specific attribute of parallel files. Therefore, when the input file format is not parallel, `ivers` = 1 and 2 are equivalent. Similarly, when the output format is not parallel the value of `ivers` is not relevant. FCONVERT cannot currently be used to change the version of an input BC file.

nvers

Specify the version number to convert the file to when `ivers` = 3. This is a real number consisting of the major and minor release numbers (ie. 3.05). Supported values of `nvers` are 2.31 - 3.05.

inform

Specify the format of input file. More information on the allowed file formats is given in Appendix A. Allowable values for `inform` are:

1	Unformatted parallel file
2	Unformatted plot3d (grid or q) file
3	Unformatted plot3d (grid or function) file
11	XDR parallel file
21	ASCII parallel file
22	ASCII plot3d (grid or q) file
23	ASCII plot3d (grid or function) file

The ASCII parallel format (`inform` = 21) can be used to run simulations, but ASCII I/O is very slow. Therefore those formats are intended primarily for debugging. Note that the multiple file formats (`inform` = 4, 14, 24) are no longer supported by DPLR.

inint

Specify whether an input zonal interface file is to be read or whether zonal interfaces are to be computed automatically. See below for more information on the use and format of zonal interface files. Input zonal interface information is only required when the input file is a plot3d grid file. This is because all zonal interface information is automatically stored in the output parallel grid file. Therefore zonal interface files are not required for the conversion of restart, BC, or radiation files, nor are they required for the conversion of a parallel grid file. See the following section for more information on zonal interface files. If a zonal interface file is specified as input when it is not required it will be silently ignored. Allowable values of `inint` are:

0	Do not read input interface file
1	Read input interface file
2	Auto-detect full face interfaces ONLY
3	Auto-detect all interfaces (fast method)
4	Auto-detect all interfaces (accurate method)

`idummy`

Specify whether input file contains dummy cells. DPLR is a finite volume code, and as such requires “dummy” or “ghost” cells at the boundaries of the computational grid. The flow values stored in these cells are used to compute high-order fluxes across grid boundaries. The coordinates of these dummy cells are used in the computation of the viscous fluxes. See the Section XX of the DPLR Reference Manual for more information. By default DPLR will automatically generate dummy cells as necessary at runtime, and it is not necessary for the user to consider them during grid generation. In this case, simply set `idummy` = 0. However, there are certain (rare) instances when the user may wish to generate their own grid dummy cell coordinates rather than allow DPLR to generate them. In this case the user should generate a grid file containing a single plane of dummy cells along each of the six faces in a 3D problem (four faces for 2D/Axisymmetric problems), transforming an *il* × *jl* × *kl* grid file into (*il*+2) × (*jl*+2) × (*kl*+2). Parallel files always contain dummy cells, and the setting of `idummy` is not important if the input file is parallel. When converting function files into restart files using `idummy` = 1 is optional, but is preferred in order to maintain proper boundary conditions at solid surfaces. Finally, note that input dummy cells will be discarded if mesh sequencing is enabled (`imseq` = 1). Allowable values of `idummy` are:

0	Input file does not contain dummy cells
1	Input file contains dummy cells

`nborig`

Specify the number of master blocks in the file. `nborig` is equivalent to the actual number of blocks in the input file, unless a recompose is to be performed (`iaction` = 3), in which case `nborig` is the final number of blocks after the recompose has been completed. See Section XX for more information on file recomposition.

ouform

Specify the format of the output file. More information on file formats is given in Appendix A. Allowable values of `ouform` are the same as for `inform` above. In addition, setting `ouform` = 0 suppresses generation of an output file, which can be useful for debugging. Note again that DPLR2D and DPLR3D cannot read or write plot3d files directly; they must be converted to one of the parallel formats before they are used. `ouform` = 11 (XDR parallel file) is the preferred format for files that are to be read into DPLR, and should be used whenever possible.

ouint

Specify whether an output zonal interface file is to be written. See below for more information on the use and format of zonal interface files. Output zonal interface files are not necessary when one of the parallel formats are selected for output, since all zonal interface information is stored in the parallel file itself. However, output zonal interface files are important for plot3d grid output, and may be useful for debugging or informational purposes in other cases. Allowable values of `ouint` are:

0	Do not write output interface file
1	Write output interface file
2	Write output interface file including dummy cells
11	Write output interface file including edges
12	Write output interface file including dummy cells and edges

Note that `ouint` = 2, 11, and 12 are provided mainly for debugging purposes, and should not generally be selected by users.

odummy

Specify whether output file contains dummy cells. The appropriate setting of `odummy` is nearly always zero. Parallel files always contain dummy cells (or at least placeholders for them), and the input value of `odummy` is ignored if the output file is parallel. FCONVERT does not attempt to create dummy cells; therefore it is not permissible to output dummy cells unless the input file already contains them (`idummy` = 1). Allowable values of `odummy` are:

0	Output file does not contain dummy cells
1	Output file contains dummy cells

One of the only uses for `odummy` = 1 is for debugging purposes, when a user wishes to view the dummy cells created by DPLR in a parallel grid file.

ncedge

Specify which edge and corner interfaces should be generated. This will be explained in more detail below. Allowable values of `ncedge` are:

0	Do not compute and edge and corner interfaces
1	Compute all edge and corner interfaces
2	Compute only edge/corner interfaces created by decomposition

This flag is intended for developers only and should be set to 1 by other users.

imseq

Specify whether mesh sequencing is to be performed. Mesh sequencing allows the user to coarsen the file in one or more of the computational (*ijk*) directions by specifying sequencing factors for each grid block, using the `(ijk)seq` flags. See below for more information. Mesh sequencing can occur simultaneously with file format changes or parallel decomposition. Allowable values of `imseq` are:

0	Do not sequence the file
1	Sequence according to the values of `(ijk)seq`
2	Sequence all blocks using `(ijk)seq` values
-2	Unsequence a restart file

iscale

Instructs FCONVERT to scale the input grid file by a constant multiplicative factor (`sfact`) before output. This is applicable only when the input is a grid file (`ifile` = 1). Grid scaling can be performed in conjunction with any of the allowable actions and with mesh sequencing. Allowable values of `iscale` are:

0	Do not scale input grid file
1	Scale input grid file by `sfact`

If `iscale` is set to 1 for any file type other than a grid file its value will be silently reset to zero by FCONVERT.

sfact

Specify the multiplicative scale factor to use when `iscale` = 1.

`nbreak`

Specify the number of blocks to decompose the input file into when `iaction` = 2. See below for more information on parallel decomposition.

`ibrk,jbrk,kbrk`

Specify decomposition factors in the *i*-, *j*-, and *k*-directions to be used when `iaction` = 1. One line of decomposition factors is required for each block in the input file. However, when parallel decomposition is not being performed, setting `ibrk` = –1 on the first line tells the code not to read additional block decomposition records. See below for more information on parallel decomposition.

`iseq,jseq,kseq`

Specify sequencing factors in the *i*-, *j*-, and *k*-directions to be used when `imseq` = 1, 2, or –2. One line of sequencing factors is required for each block in the input file, unless `imseq` = 2. In this case the code assumes that all blocks are to be sequenced by the same factor, and only one line of sequencing factors is required, regardless of the number of grid blocks. When sequencing is not being performed, setting `iseq` = 0 on the first line tells the code not to read additional block sequencing records. See below for more information on file sequencing.

`iname`

Specify the input file name. This is the file that will be processed by FCONVERT. The filename should be surrounded by single or double quotes, and can be specified with either a relative or an absolute path. Note that the file suffix is optional; FCONVERT will assume the default suffix for the specified file type if not entered. Default suffixes are for each file type are given in Section XX.

`xname`

Specify the input interface file name (if any). The filename should be surrounded by single or double quotes, and can be specified with either a relative or an absolute path. Note that the file suffix is optional; FCONVERT will assume the default suffix ("`.inter`") if not entered.

oname

Specify the output file name. The filename should be surrounded by single or double quotes, and can be specified with either a relative or an absolute path. Note that the file suffix is optional; FCONVERT will assume the default suffix for the specified file type if not entered. Default suffixes for each file type are given in Section XX. Note that if the output filename (with extension) is the same as the input filename, the input file will be overwritten, which is not typically a desired result. If an output interface file is requested, the suffix "`.inter`" will be appended to the prefix specified by `oname`.

nsin

Specify the number of chemical species. This is only read if the user is attempting to create a restart file from a plot3d file. It is necessary to provide this information in the input deck since FCONVERT cannot determine it from the plot3d file itself. See Section XX for a discussion of how to convert a plot3d file into a restart file.

nerin

Specify the number of unique rotational temperatures (energy conservation equations). This is only read if the user is attempting to create a restart file from a plot3d file. It is necessary to provide this information in the input deck since FCONVERT cannot determine it from the plot3d file itself. See Section XX for a discussion of how to convert a plot3d file into a restart file.

nevin

Specify the number of unique vibrational temperatures (energy conservation equations). This is only read if the user is attempting to create a restart file from a plot3d file. It is necessary to provide this information in the input deck since FCONVERT cannot determine it from the plot3d file itself. See Section XX for a discussion of how to convert a plot3d file into a restart file.

necin

Specify the number of unique electronic temperatures (energy conservation equations). This is only read if the user is attempting to create a restart file from a plot3d file. It is necessary to provide this information in the input deck since

FCONVERT cannot determine it from the plot3d file itself. See Section XX for a discussion of how to convert a plot3d file into a restart file.

`ntbin`

Specify the number of turbulence variables. This is only read if the user is attempting to create a restart file from a plot3d file. It is necessary to provide this information in the input deck since FCONVERT cannot determine it from the plot3d file itself. See Section XX for a discussion of how to convert a plot3d file into a restart file.

Zonal Interfaces

Zonal interfaces describe how one or more block faces of a grid file abut, or share points, with other blocks. Each contiguous region of abutment is called a zonal interface, or a zonal boundary. Definitions of these zonal interfaces must be provided as input to DPLR (`inint` = 1) or computed directly by FCONVERT (`inint` = 2-4) so that information can be passed correctly between blocks during the CFD computation. Zonal interface data is passed to DPLR via an input zonal interface file, described in the next section. At this time DPLR only supports point-matched structured grids, and thus a zonal interface indicates a region where two grid blocks share the same grid points.

Zonal interfaces frequently occur in multiblock grids, but can also occur in single block grids. For example, an O-grid about an airfoil section has a zonal interface where the grid is "stitched" together at the trailing edge. If the (3D) grid contains a singular axis, a zonal boundary is required to describe the mapping of "dummy" cells across the axis. Finally, zonal interfaces can be used to simulate problems with periodic boundary conditions. In this case the zonal interface is specified to enforce periodicity in the computational domain, and does not relate faces that are physically connected. In any case, DPLR uses the zonal boundary information to ensure that data are mapped correctly across grid blocks in order to maintain a uniform high-order extrapolation of the Euler and Navier-Stokes fluxes. A single layer of dummy cell *coordinates* is required across all zonal boundaries to ensure that the viscous fluxes are computed correctly. In addition, two layers of dummy cells are necessary across all zonal boundaries to ensure uniform second or third-order accuracy. DPLR automatically creates two layers of dummy cells from the input zonal boundary information.

Zonal interface information is considered to be a property of the input grid in DPLR. Therefore all zonal interface information is stored in the header of the parallel grid file during processing by FCONVERT. This information can be extracted directly from the parallel grid file if further processing is required. Because of this, input zonal interface files are only required when processing plot3d grids which contain one or more zonal boundaries.

Ways to Determine Zonal Interface Information

Some form of zonal interface data is required by FCONVERT whenever one or more zonal interfaces exist in the grid file to be processed. Prior to V3.04.1 the only way to determine these interfaces was to provide FCONVERT with an input interface file, as described in the following sections. This is done by specifying `inint` = 1 in the FCONVERT input deck. Starting with V3.04.1 of the package FCONVERT is also able to compute these interfaces automatically under most circumstances. This is done by specifying `inint` = 2-4 in the input deck when processing an input grid file. This is described in the section "Automatic Interface Computation". In any case, some understanding of the format and logic behind the interface data in DPLR is useful to understand.

Interface File Format

A zonal interface file is required as input to FCONVERT whenever one or more zonal interfaces exist in the grid file to be processed if `inint` = 1 in the FCONVERT input deck. In this case the user must specify the name of the interface file with `xname`. The interface file is always ASCII, and describes how the block faces of a grid file abut, or share points, with other blocks.

Zonal interface files are generally created by hand during the grid generation process. This can be a time consuming activity for problems with many grid blocks. However, some commercial grid generation tools are capable of automatically generating interface information in a format that is readable by the commercial CFD code *GASP® Version 3*. For this reason, a tool (***zbconvert***) is included with the DPLR software package that can convert zonal interface information from *GASP® Version 3* to DPLR format. See Appendix U for more information. The format of a simple zonal interface file is shown here:

```
 ZONAL BOUNDARY INFORMATION
 Cell Matching - No dummy cells
 -----------------------------------------
   zvers izdum
   3.05    0

   nblk ninta nintc
    3    2    0

 -----------------------------------------
Zonal Boundary #  1
   nz nface ndr1 nst1 nen1 ndr2 nst2 nen2
    1    2    3    1   30    2    1   16
    2    1    3    1   30    2    1   16

 -----------------------------------------
Zonal Boundary #  2
   nz nface ndr1 nst1 nen1 ndr2 nst2 nen2
    2    2    3    1   30    2    1   16
    3    1    3    1   30    2    1   16


 =========================================
```

The first four lines of the file are comments and are not read by the code. The remainder of the file is broken into a header section, followed by zonal interface specifications. Flags in the header section are:

`zvers`

Specify the version number of the interface file. This is used by FCONVERT to automatically upconvert older interface files when they are read, and thus assures full backward compatibility. `zvers` is a real number consisting of the major and minor release numbers (ie. 3.05).

`izdum`

Specify whether dummy cells are accounted for in the interface file. This option is meant for debugging purposes; in general non-developers should set `izdum` = 0. If FCONVERT is given an input file with `izdum` = 1 it will automatically strip the dummy cell information before processing the file.

0	Input file does not include dummy cells
1	Input file includes dummy cells

`nblk`

Specify the number of blocks in the grid. `nblk` must be consistent with the number of blocks in the input grid file or the program will abort.

`ninta`

Specify the number of normal (face) zonal interfaces in the file.

`nintc`

Specify the number of corner/edge zonal interfaces in the file. This flag is also present for debugging purposes, and should not be set to a non-zero value by

> non-developers. See Appendix F for a description of corner/edge interfaces. If FCONVERT is given an input file with `nintc` > 0 it will automatically strip the corner/edge zonal interface information before processing the file.

Following the header section is the body of the file, which consists of a zonal boundary specification block for each zonal interface in the file. If there are fewer than `ninta` zonal boundary specification blocks a runtime error will occur. If there are more than `ninta` zonal boundary specification blocks the extra information will be silently ignored. Each block consists of a pair of data lines that completely describe extent and range of the point-matched zonal interface.

`nz` specifies the grid blocks that define the common face. In the example file above, zonal boundary number one (ZB #1) involves grid blocks #1 and #2 of a three block file. `nface` specifies the block faces that abut. Block faces are assigned a number between 1 and 6, where:

1	*imin* face
2	*imax* face
3	*jmin* face
4	*jmax* face
5	*kmin* face
6	*kmax* face

If the grid is for a 2D or axisymmetric problem, `nface` must lie between 1 and 4, since such problems are assumed to lie in the *ij*-plane.

In ZB #1 above, it can be seen that the *imax* face of block #1 abuts the *imin* face of block #2. These flags would be sufficient to define the zonal interface if the *entire imax* face of block #1 was matched to the *entire imin* face of block #2. However, DPLR allows for partial face interfaces (sometimes called subfacing). In fact, a single grid face in one block can in general match with multiple faces in multiple blocks in DPLR. Therefore it is also necessary to define the extent of the interface in the *ijk* directions. The value of `nface` for each side of the interface defines a plane in computational space; the extent of the abutment must be described using extents of the two variables in that plane. For example, in ZB #1 above, block #1 has `nface` = 2, which implies an *imax* face, or a *jk*-plane. Therefore the extent must be specified in the *j*- and *k*-directions.

The extent direction is specified using the `ndir1` and `ndir2` flags. `ndir1` and `ndir2` are specified as a number from one to three, where:

1	*i*-direction
2	*j*-direction
3	*k*-direction

The extent directions can be specified in any order, but must correspond to the plane defined by the setting of `nface`. In the example above, if `ndir1` is set to 2, then `ndir2` must be set to 3, and vice versa. The range in each direction is then defined using the `nst1 - nen1` and `nst2 - nen2` flags, where `nst1` and `nen1` define the range for `ndr1`, and `nst2` and `nen2` define the range for `ndr2`. Note that the range is defined by the number of cell centers that abut, rather than grid lines. It is important that extent definitions for block #1 match those for block #2, so that the interface region is uniquely defined.

Zonal Interface Examples

This sounds a lot harder than it really is, but can be made much clearer with a couple of 2D examples. In the simplest case (Example #1), shown schematically in Fig. 1, block #1, with grid point dimensions 6 × 8 (or 5 × 7 cells), abuts block #2, with dimensions 8 × 6. From the figure, we see that the *imax* face of block #1 fully abuts the *jmax* face of block #2. The zonal interface for this case can be specified as:

```
Zonal Boundary #  1
   nz nface ndr1 nst1 nen1 ndr2 nst2 nen2
    1    2    2    1    5    3    1   1
    2    4    1    1    5    3    1   1
```

The extent range is defined to be $j = 1 - 5$ in block #1 matching with $i = 1 - 5$ in block #2, where again the starting and ending points of the range are determined by cells (as labeled in Fig. 1) rather than grid points. Since this example is two-dimensional, the range in the *k*-direction is specified to be from 1 to 1.

Example 2 is shown schematically in Fig. 2. In this example, the two blocks have the same dimension, but this time the *imax* face of block #1 partially abuts the *imax* face of block #2. The zonal interface for this case can be specified as:

```
Zonal Boundary #  1
   nz nface ndr1 nst1 nen1 ndr2 nst2 nen2
    1    2    2    1    3    3    1   1
    2    2    2    3    1    3    1   1
```

The extent range in this example is defined to be $j = 1 - 3$ in block #1 matching with $j = 3 - 1$ in block #2. Note that since j runs in the opposite direction in the two blocks it is necessary that `nst1` > `nen1` for one of the range specifiers in order to define the point matching of this interface correctly.

Example 3 is shown schematically in Fig. 3. In this example, an L-shaped block #1 abuts block #2. In this case the *imax* face of block #1 is subfaced; it abuts two separate faces of block #2. The zonal interfaces for this case can be specified as:

```
Zonal Boundary #  1
   nz nface ndr1 nst1 nen1 ndr2 nst2 nen2
    1    1    2    1    5    3    1   1
    2    2    2    1    5    3    1   1

---------------------------------------
Zonal Boundary #  2
   nz nface ndr1 nst1 nen1 ndr2 nst2 nen2
    1    4    1    1    5    3    1   1
    2    2    2    6   10    3    1   1
```

By default all zonal interfaces are assumed to be two-way. This means that flux information is passed in both directions across the zonal boundary. However, it is also possible to pass information in a single direction only. The most common use of this option is internal to the code; degenerate axes and certain edge or corner interfaces are by their nature single-direction. These cases are handled automatically by FCONVERT and do not need to be discussed here. See Appendix F for further discussion of edge and corner interfaces.

In order to specify a single direction interface, simply put a negative sign in front of the grid block number that *receives* the information. As an example, assume that the zonal interface defined in Example #1 (Fig. 1) is now a one-way interface, with data passing from block #2 to block #1, but not the other way. In this case block #1 is the data receiver, and the resulting zonal interface specification would look like this:

```
Zonal Boundary #  1
   nz nface ndr1 nst1 nen1 ndr2 nst2 nen2
   -1    2    2    1    5    3    1   1
    2    4    1    1    5    3    1   1
```

Finally, it is important to note that the order in which the interface lines are written, and the order in which the ranges are specified, is not important as long as the interface is correctly defined. For example, the previous interface specification could also have been written as:

```
Zonal Boundary #  1
   nz nface ndr1 nst1 nen1 ndr2 nst2 nen2
    2    4    3    1    1    1    5   1
   -1    2    3    1    1    2    5   1
```

However, the following specification would be incorrect:

```
Zonal Boundary #  1
   nz nface ndr1 nst1 nen1 ndr2 nst2 nen2
    2    4    1    5    1    3    1   1
   -1    2    3    1    1    2    5   1
```

since the extent and range specifiers now attempt to map the *i*-direction in block #2 with the *k*-direction in block #1, and the *k*-direction in block #2 with the *j*-direction in block #1. This zonal interface specification would result in a run-time error in FCONVERT, since it does not specify an appropriately point-matched interface.

All of the above examples discuss the definition of face interfaces. However, DPLR recognizes three distinct types of zonal interfaces: face interfaces, edge interfaces, and corner interfaces. Face interfaces must be input to FCONVERT via an input interface file as described above. Edge and corner interfaces are also necessary for the computation of full viscous fluxes. However, edge and corner interfaces are computed automatically within FCONVERT rather than entered as input. While it is not necessary to understand the role of edge and corner interfaces in order to use FCONVERT, a description is provided for the interested reader in Appendix F.

Additional examples of zonal interface files and their use by FCONVERT can be found in the sample cases included with the software distribution; the user is encouraged to go through the provided examples carefully.

Automatic Interface Computation

There are three options for letting FCONVERT compute zonal interface data automatically when input multiblock grids are processed. These options are discussed here in order of increasing accuracy (and computational time requirement).

In many cases it can take some time to compute zonal interface information for a large complex grid; therefore it is recommended that the user write out the resulting interface file (`ouint` = 1) the first time this action is performed. Future applications of FCONVERT can then be run using `inint` = 1, saving considerable processing time.

`inint = 1`

Read an interface file as input. This option is described in previous sections above.

`inint = 2`

Rapid detection of full-face zonal interfaces. This is the quickest detection option, and it will typically execute almost immediately on a modern machine. This option works by comparing the centroid of each block face (where the centroid is computed by averaging all cells in that face). Index directions of the two faces can be arbitrary as long as the centroids of a face pair are within the tolerance. As this method detects full-face interfaces only, this option should only be used if it is known that the input grid does not contain sub-face interfaces (see previous section for definitions).

`inint = 3`

This option detects both full-face and sub-face interfaces, and this is the option that is recommended for most cases. Speed of this option will be moderate. Although the observed speed of the algorithm is heavily problem dependent, 3D test grids have shown that this option will take on the order of one minute per million grid cells to complete on a modern Pentium 4 or Opteron processor. This option works by analyzing all edge cells of all block faces for interfaces. For example, this option will examine the *imin*, *imax*, *jmin*, and *jmax* lines on the *kmax* face of a block. A typical block is shown in Fig. 4, where edge cells that are checked are highlighted in red. Thus, option `inint` = 3 and `inint` = 4 should produce identical results for 2D/axisymmetric problems, but in 3D each zonal interface must touch at least one edge cell of at least one block. The matching cell may be located on either the edge or the interior of another block face. The only type of interface that will not be detectable using the `inint` = 3 option is a case where an interior sub-face of a 3D block face touches another interior sub-face of the same block. An example of this arrangement is shown in Fig. 5. Such an arrangement is not typically used in grid generation, so cases where this interface detect option does not work will be extremely rare.

`inint = 4`

This option provides accurate detection of all zonal interfaces. This option detects both full-face and sub-face interfaces using a rigorous but slow search algorithm, where every single exposed face cell is compared to every other exposed face cell. Although the observed speed of the algorithm is heavily problem dependent, 3D test problems have shown that this option will take on the order of fifteen minutes per million grid cells to complete on a modern Pentium 4 or Opteron processor. This option will find the interior-to-interior zonal boundaries that `inint` = 3 cannot (shown in Fig. 5), but cases requiring the use of this option will be infrequent.

As stated previously, DPLR requires point-matched interfaces at this time. Because of this, accurate detection of zonal interfaces requires that the interfaces be truly point matched. FCONVERT computes a tolerance factor by taking the minimum of the length, width, and height of all boundary cells in each block. This tolerance factor is then used to determine if two face cells are in the same spatial location. The idea behind this technique is that the spatial distance between zonal interface pair of cells must be small when compared to the size of the boundary cells in the blocks.

Additionally, intra-zonal interface boundaries (singularity axes, self-closing blocks, etc.) will be detected using any of the detection options `inint` = 2-4. The `inint` = 2 option has the same criterion that the entire face of the block must coincide just as with block-to-block type interfaces.

Some grid generation packages and post-processors have a tendency to introduce roundoff error in the *xyz* grid coordinates that can result in the points on either side of the interface being slightly different. FCONVERT has a built in tolerance factor to determine when two slightly different points are likely the same, but this is not foolproof. FCONVERT outputs the tolerance of each interface found in the input grid at run-time; a large number of non-zero tolerance factors could indicate a "sloppy" grid file and may imply that some interfaces are missed. At this time there is no way to determine whether all interfaces have been detected other than to either (1) compute them all by hand as a check case, or (2) run the resulting case and look for mismatched interfaces in the resulting solution.

Parallel Decomposition

The most common use of FCONVERT is to decompose an input grid file for parallel execution. When using DPLR it is important to remember that it is a distributed-memory parallel code. All grid blocks are computed simultaneously rather than sequentially, and multi-block information transfer is handled through MPI data constructs; therefore it is necessary to run on *at least* as many processors as blocks in the original computational grid. Running on more processors than master grid blocks is often advantageous, since the largest blocks can then be split (decomposed) into smaller pieces, increasing computational efficiency and decreasing turnaround time. This decomposition, if required, is performed using FCONVERT.

The "ideal" number of processors to use for a given job is a matter of personal preference; it is generally a function of the total number of processors that are available and the number that are are necessary to achieve a reasonable load balance. However, once the desired number of processors to use during the run has been selected, the input grid file must be decomposed into one block per processor. This is accomplished by setting `iaction` = 1 or 2 in the input deck.

Decomposition Strategies

When `iaction` = 1, the user specifies how each block in the input file is to be decomposed using the `ibrk`, `jbrk`, and `kbrk` decomposition factors. One set of decomposition factors is required for each block in the input file. A decomposition factor of *n* implies that the block should be broken *n* times in that direction. For example, a decomposition record of

```
Decomposition information for each master block
ibrk    jbrk    kbrk
  2       3       1
```

indicates that the original block should be split into six by breaking it into two equal pieces in the *i*-direction and into three equal pieces in the *j*-direction. If the number of computational cells in a given direction is not evenly divisible by the selected decomposition factor the remainder will be evenly distributed among the blocks. Setting `iaction` = 1 allows the user to control the way that the problem is decomposed for parallel execution, which can have significant advantages, as discussed below.

When `iaction` = 2 the user simply specifies the desired number of output blocks using the `nbreak` flag. FCONVERT will determine a parallel decomposition that divides the original file into `nbreak` output blocks. The blocks will be broken such that load balance is maximized. This means that FCONVERT will attempt to make all blocks as close as possible to the same size. In addition, FCONVERT will attempt to make the blocks as close to cubes as possible by breaking first in the direction(s) with the most points.

While `iaction` = 2 requires less input from the user, it is generally *not* the preferred method to perform a parallel decomposition. This is because DPLR is by default a line relaxation code that solves the Navier-Stokes equations through a series of block-tridiagonal matrix factorizations. This method converges most rapidly when the problem has not been decomposed in the body-normal direction. See Section XX of the DPLR Reference Manual for more information. However, FCONVERT does not "know" which direction(s) are body normal when performing a decomposition. Therefore, it is generally preferable to manually specify block decomposition factors for each block using `iaction` = 1 and the `ibrk`, `jbrk`, and `kbrk` flags as discussed above. In this strategy block decomposition factors can be user-specified so as to minimize or eliminate breaks in the body-normal direction.

As an example, consider an input grid file that consists of two blocks. The plot3d header record for this case is shown here:

```
2
17  33  129
65  65  129
```

Block #1 consists of 65,536 grid cells (16 × 32 × 128), while block #2 consists of 524,288 cells (64 × 64 × 128). Assuming that `iaction` = 2 is selected with `nbreak` = 7, a portion of the descriptive output for this run is shown here:

```
Input Block  1 size: il = 16; jl =  32; kl = 128  ( 65536 cells)
Input Block  2 size: il = 64; jl =  64; kl = 128  (524288 cells)

 Largest block is:
      nb =   2; original block =   2
      il =  64; jl =  64; kl =  128


 Read input interface file neptune.inter
 Found   3 valid zonal interface blocks in   2 block grid file


 Decomposing block   1 into   1:   ibrk=  1 jbrk=  1 kbrk=  1
 Decomposing block   2 into   6:   ibrk=  2 jbrk=  1 kbrk=  3
 -----------------------------------------
                  creating   7 total blocks

   7 Blocks;  Max load imbalance =  25.58%

 Output Block 1 size: il = 16; jl = 32; kl = 128  ( 65536 cells)
 Output Block 2 size: il = 32; jl = 64; kl =  43  ( 88064 cells)
 Output Block 3 size: il = 32; jl = 64; kl =  43  ( 88064 cells)
 Output Block 4 size: il = 32; jl = 64; kl =  43  ( 88064 cells)
 Output Block 5 size: il = 32; jl = 64; kl =  43  ( 88064 cells)
 Output Block 6 size: il = 32; jl = 64; kl =  42  ( 86016 cells)
 Output Block 7 size: il = 32; jl = 64; kl =  42  ( 86016 cells)
```

As can be seen, FCONVERT decomposed block #2 into six nearly equal pieces while leaving block #1 unaltered. The resulting load imbalance was 25%, indicating that the largest block has about 25% more points than the smallest. This is the most load-balanced solution for `nbreak` = 7, but it may not be the most desirable way to split the problem. If for example the *k*-direction is body-normal for this problem it would be preferable to select a decomposition that does not break the problem in the *k*-direction. This can be accomplished by setting `iaction` = 1 and using the block decomposition flags to specify the desired decomposition. For this example, the following decomposition would be preferred:

```
Decomposition information for each master block
ibrk    jbrk    kbrk
  1       1       1
  3       2       1
```

A portion of the FCONVERT output for this run is shown here:

```
Input Block 1 size: il =  16; jl =  32; kl = 128  ( 65536 cells)
Input Block 2 size: il =  64; jl =  64; kl = 128  (524288 cells)

 Largest block is:
      nb =   2; original block =   2
      il =  64; jl =  64; kl =  128


 Read input interface file neptune.inter
 Found   3 valid zonal interface blocks in   2 block grid file

 Decomposing block   1 into   1:   ibrk=  1 jbrk=  1 kbrk=  1
 Decomposing block   2 into   6:   ibrk=  3 jbrk=  2 kbrk=  1
 -----------------------------------------
```

```
                      creating   7 total blocks

   7 Blocks;  Max load imbalance =  27.27%

 Output Block 1 size: il = 16; jl = 32; kl = 128  ( 65536 cells)
 Output Block 2 size: il = 22; jl = 32; kl = 128  ( 90112 cells)
 Output Block 3 size: il = 21; jl = 32; kl = 128  ( 86016 cells)
 Output Block 4 size: il = 21; jl = 32; kl = 128  ( 86016 cells)
 Output Block 5 size: il = 22; jl = 32; kl = 128  ( 90112 cells)
 Output Block 6 size: il = 21; jl = 32; kl = 128  ( 86016 cells)
 Output Block 7 size: il = 21; jl = 32; kl = 128  ( 86016 cells)
```

This decomposition has a slightly larger load imbalance (27% vs. 25%), but would likely converge faster when run in DPLR, assuming that the *k*-direction is body-normal. Note that this solution is not unique; there are several other possible decompositions that would achieve the same result.

Finally, when `iaction` = 10, FCONVERT will generate an output file with the same number of blocks as the input file; ie no further decomposition will be performed. The same result could be achieved either by

1) setting `iaction` = 1 and all `ibrk`, `jbrk`, `kbrk` flags to be one, or
2) setting `iaction` = 2 and `nbreak` equal to `nborig`.

FCONVERT will automatically compute all additional face, edge, and corner zonal interfaces created by the specified parallel decomposition. In addition, if the input grid contains one or more zonal interfaces, these will be automatically decomposed along with the grid file. This information will be written to the output grid file header if one of the parallel formats is requested. In addition, the user can request that the resulting zonal interface file be output for informational purposes by setting `ouint` = 1, 11, or 12. Decomposing a file in multiple directions can create a large number of output zonal interfaces, particularly when edge and corner interfaces are considered. Since each zonal interface represents a message that must be constructed and sent via MPI send and receive calls each iteration during the CFD solution, it is generally a good idea to keep decompositions as simple as possible. For example, if it is desired to run a single block 3D problem on eight processors, the simplest decomposition would be to break the problem into eight blocks in a single coordinate direction, which would generate 7 face interfaces and zero edge or corner interfaces. An alternate strategy would be to break into 4 × 2 × 1 blocks, which would generate 10 face interfaces and 6 edge interfaces, for a total of 16. The most complex decomposition would be 2 × 2 × 2 blocks. This strategy would generate 12 face interfaces, 12 edge interfaces, and 4 corner interfaces, for a total of 28. Although each of these strategies are allowed, clearly the first strategy would generate the least message-passing traffic during run-time, and would likely result in the most time-efficient solution.

Results of Decomposition

The actual action taken by FCONVERT during a file decomposition depends on the output file format. If the output format selected is plot3d, the input file will be physically split into multiple blocks and written as a multi-block file. Note that plot3d file output is provided mainly for informational purposes, since DPLR cannot read plot3d files directly.

Whenever a parallel output file format is selected, the resultant file retains information about the original block structure. Another way of thinking of this is that FCONVERT distinguishes between "virtual" blocks, which are generated purely to facilitate parallel execution, and "physical" blocks, which are a fundamental property of the input grid. On the other hand, the user interfaces to DPLR2D, DPLR3D, and POSTFLOW deal only with physical blocks – "virtual" blocks are automatically converted to and from physical blocks as required during program execution. Therefore when setting up a problem to run in DPLR only the physical ("master") block structure of the problem is important. If a two-block grid is decomposed into `nblk` "virtual" blocks to run in parallel, the problem is set up for DPLR as a two-block problem, regardless of the actual value of `nblk`. This means that boundary conditions, numerical models, etc. are only specified for the two master blocks. DPLR will automatically convert this information to the "virtual" values at runtime. Similarly, when the solution is post-processed by POSTFLOW, it is treated as a two-block problem, regardless of the actual number of processors that were used. This strategy greatly simplifies the preparation, execution, and post-processing overhead required for parallel jobs.

The output of FCONVERT provides information on the physical block structure, and includes physical block sizes, which are required for setting up the DPLR input deck. A portion of the output from FCONVERT for the sample problem of the previous section is shown here:

```
Summary (grid dimensions for CFD input deck):
 Hardwired to run on   7 processors

      Block   1; nx =  16; ny =  32; nz = 128
      Block   2; nx =  64; ny =  64; nz = 128
```

The summary information states that the problem has been decomposed (or "hardwired") for execution on seven processors, but there are only two physical blocks that must be considered during the problem setup. Using this strategy, once a DPLR or POSTFLOW input deck has been created for a given problem, the *same* input deck can be used regardless of actual the number of processors employed in the solution. The user is never required to visualize or work with the parallel decomposition of the problem.

If the output format selected for the decomposed file is one of the parallel archival formats (`ouform` = 1, 11, 21), a single output file will be created. A parallel archival file actually contains only as many blocks as the original, but additional information is written to the file header to tell DPLR how to perform the appropriate decomposition at run-time. This "virtual" decomposition information is written only to the grid file header. Therefore parallel archival restart files never need to be decomposed or recomposed. Once a parallel archival grid file has been created, it is considered to be “hardwired” for a given number of processors. This will be reflected in the output messages produced when FCONVERT is run. If the user later wishes to run or restart the problem on a different number of processors, the grid file can simply be decomposed again. FCONVERT will strip out the header information, decompose the master blocks as desired, and write the new header information into the file.

Testing for Load Balance

It is frequently the case that many processors are available for the run, but it is desired to choose a number that maximizes the load balance (and therefore computational efficiency). It is possible to test the load balance for a series of possible decompositions with FCONVERT. This is done by setting `iaction` = 0, and setting `nbreak` to the maximum number of blocks desired. FCONVERT will then loop over all possible output block numbers from the number of input blocks to the value of `nbreak`, and output the most load balanced way to decompose into that number of output blocks. For the example above, if `iaction` = 0 and `nbreak` = 10 the following output will be generated:

```
 Decomposing block   1 into   1:   ibrk=  1 jbrk=  1 kbrk=  1
 Decomposing block   2 into   1:   ibrk=  1 jbrk=  1 kbrk=  1
   2 Blocks;  Max load imbalance =  87.50%

 Decomposing block   1 into   1:   ibrk=  1 jbrk=  1 kbrk=  1
 Decomposing block   2 into   2:   ibrk=  1 jbrk=  1 kbrk=  2
   3 Blocks;  Max load imbalance =  75.00%

 Decomposing block   1 into   1:   ibrk=  1 jbrk=  1 kbrk=  1
 Decomposing block   2 into   3:   ibrk=  1 jbrk=  1 kbrk=  3
   4 Blocks;  Max load imbalance =  62.79%

 Decomposing block   1 into   1:   ibrk=  1 jbrk=  1 kbrk=  1
 Decomposing block   2 into   4:   ibrk=  2 jbrk=  1 kbrk=  2
   5 Blocks;  Max load imbalance =  50.00%

 Decomposing block   1 into   1:   ibrk=  1 jbrk=  1 kbrk=  1
 Decomposing block   2 into   5:   ibrk=  1 jbrk=  1 kbrk=  5
   6 Blocks;  Max load imbalance =  38.46%

 Decomposing block   1 into   1:   ibrk=  1 jbrk=  1 kbrk=  1
 Decomposing block   2 into   6:   ibrk=  2 jbrk=  1 kbrk=  3
   7 Blocks;  Max load imbalance =  25.58%

 Decomposing block   1 into   1:   ibrk=  1 jbrk=  1 kbrk=  1
 Decomposing block   2 into   7:   ibrk=  1 jbrk=  1 kbrk=  7
```

```
   8 Blocks;  Max load imbalance =  15.79%

 Decomposing block   1 into   1:   ibrk=  1 jbrk=  1 kbrk=  1
 Decomposing block   2 into   8:   ibrk=  2 jbrk=  2 kbrk=  2
   9 Blocks;  Max load imbalance =   0.00%

 Decomposing block   1 into   1:   ibrk=  1 jbrk=  1 kbrk=  1
 Decomposing block   2 into   9:   ibrk=  3 jbrk=  1 kbrk=  3
  10 Blocks;  Max load imbalance =  13.87%

           Finished with Load Balance Check
```

From this we see that a perfectly load balanced solution is possible if the problem is decomposed to run on nine processors.

Single Block Input Files

In general parallel decomposition must be performed by FCONVERT. However, in the special case of a single block grid with no zonal interfaces, DPLR2D and DPLR3D can perform parallel decomposition at runtime. In this case the input grid file can simply be converted to parallel archival format (`iaction` = 10). The resulting file can be run on any number of processors without further processing by FCONVERT.

Parallel Recomposition

FCONVERT can also be used to "recompose" a grid file that was previously decomposed. This action is rarely used in practice, since as discussed in the previous section, when the output file is written in one of the archival parallel formats (`ouform` = 1, 11, or 21) any selected decomposition is virtual; the output file merely contains header information that instructs DPLR2D or DPLR3D how to properly decompose the file at runtime. Therefore it is never necessary to recompose archival parallel files. In addition, since restart, BC, and radiation files are never decomposed in the first place, this option can only be used with grid files. If `iaction` = 3 is specified with an archival parallel file as input all that will occur is the virtual decomposition information will be stripped from the file header. In addition, boundary condition and radiation files are never decomposed for parallel execution, and thus cannot be "recomposed."

When `iaction` = 3 is selected it is also necessary to specify the number of blocks in the recomposed file with the `nborig` flag. If the input file is in plot3d format is it also important to provide the input interface file `inint` = 1, which provides information about how the original grid or restart file was decomposed. While FCONVERT will recompose an input grid file, it does not attempt to recreate the zonal interface file for the recomposed problem. It is therefore important that the original interface file be saved; otherwise the file will need to be recreated after the recompose is complete.

Mesh Sequencing

DPLR does not perform any multigrid cycles, nor does it directly support mesh sequencing for convergence acceleration. This is because the line relaxation method employed is already extremely efficient, and typically there is no benefit to starting a fine-grid solution from a converged solution on a coarser mesh. However, it can occasionally be advantageous to obtain a solution on a coarser mesh (for example wake flows, or in order to perform grid convergence studies). Therefore an option is included in FCONVERT to sequence (coarsen) a grid, radiation, BC, or restart file, and create a new output file. The coarse grid simulation would then run independently of the fine grid simulation.

Mesh sequencing is accomplished by setting `imseq` = 1 or 2 in the input deck. When mesh sequencing is enabled, the user controls the level of sequencing desired in each grid block with the `iseq`, `jseq`, and `kseq` sequencing factors. One set of sequencing factors is required for each block in the input file (`imseq` = 1) or else a single set of sequencing factors is assumed to apply to all block in the grid (`imseq` = 2). A sequencing factor of *n* implies that the block should be coarsened *n* times in that direction. For example, a sequencing record of

```
Sequencing information for each master block
iseq    jseq    kseq
  2       1       3
```

indicates that the number of computational cells in the block should be reduced by a factor of two in the *i*-direction and by a factor of three in the *k*-direction. This is done by eliminating every other grid point (or computational cell) in the *i*-direction, and two out of every three points in the *k*-direction. As an example, if the original grid had dimensions of 33 × 65 × 67 grid points (32 × 64 × 66 computational cells), after sequencing by the above factor the final grid would have 17 × 65 × 23 points, or 16 × 64 × 22 cells.

It is an error to attempt to sequence a file by a factor by which it is not evenly divisible. In the example above a sequencing record of

```
Sequencing information for each master block
iseq    jseq    kseq
  3       1       3
```

would result in a runtime error, since the number of cells in the *i*-direction (32) is not evenly divisible by 3. It is also important for multiblock problems to ensure that the sequencing strategy you have chosen results in point matched coarse grids. If you have two abutting grid blocks, each with 61 × 61 × 61 grid points, and sequence block #1 by 2 in each direction and block #2 by 3 in each direction, the final grid will not be point matched across the zonal interface. This will NOT necessarily generate a runtime error in FCONVERT, but will cause problems when you try to run the resulting case.

If the input grid has zonal interface information associated with it, these data will be automatically sequenced along with the grid file. Once you have appropriately sequenced the grid file, BC file (if any), and radiation file (if any), you can set up and run the problem independently from the fine grid solution. It is also possible to sequence restart files, although this is of limited practical usefulness, since it is typically preferable to start the coarse grid solution from freestream rather than from a sequenced fine grid solution.

Upsequencing Restart Files

Finally, FCONVERT allows the user to "upsequence" a restart file, by setting `imseq` = –2. In other words, the user can make a restart file "finer" in one or all of the coordinate directions. This can be used to emulate the mesh sequencing capability in *GASP*, in which the solution from the coarse mesh is automatically used as the starting solution on the next finer mesh. As stated previously, this is usually not necessary as a convergence acceleration tactic in DPLR, but there are occasions when it may be useful. In order to employ this technique, the user would first sequence a starting grid file in order to create a coarser version. After running this coarse solution in DPLR, the user would then "upsequence" the restart file to the finer grid by setting `imseq` = –2 in the FCONVERT input deck. When `imseq` = –2, the `iseq`, `jseq`, and `kseq` flags have the opposite meaning. In other words, a sequencing record of

```
Sequencing information for each master block
iseq    jseq    kseq
  2       1       3
```

during an upsequence would turn a problem with 16 × 64 × 22 computational cells into one with 32 × 64 × 66 cells. This is done by simply replicating each point in the coarse file; no interpolation is performed. Once the restart file has been upsequenced, it can be used to initialize the fine grid simulation. Note that FCONVERT will only allow upsequencing of restart files. This is because the upsequencing process cannot return files to their original state; an upsequenced grid file would simply have extra points added algebraically, it would not retain the grid smoothness and stretching function of the original file. In the case of a restart file, the upsequencing is used only to initialize a problem, and thus interpolation errors will be washed out in the final solution.

Converting a Plot3d Function File to a Restart File

It is occasionally useful to convert a plot3d function file into restart file format. The most common uses of this feature are to import function files created by the grid adaption tool *SAGe*. This permits the user to perform a simulation, adapt the grid to the computed solution, and restart the solution on the adapted grid, rather than beginning the adapted grid simulation from freestream conditions. The conversion of plot3d function files to restart files can also be useful if it is desired to restart a solution on one grid by interpolating flow data from another solution on a different grid. As discussed previously, these types of restarts seldom provide an advantage when using line relaxation, but the capability is provided nevertheless.

When creating a restart file the input function file must contain the following variables in the following order:

$$\rho_s, u, v, (w), T, (T_R), (T_V), (T_e), (turb)$$

where ρ_s are the species densities, *u*, *v*, and *w* are the velocity components, *T* is the translational temperature, T_r is the rotational temperature, T_v is the vibrational temperature, T_{el} is the free electron temperature, and *turb* are the turbulence variables. Only those variables that are required should be included; for example *w* is not included for a two-dimensional or axisymmetric flow, and the vibrational temperature is not included for a perfect gas or thermal equilibrium flow. Since plot3d files contain no information about the types of variables that they contain it is also necessary to provide FCONVERT with some additional information. In particular, `nsin` should be set equal to the number of chemical species, and `nerin`, `nevin`, `necin` should be set equal to the number of independent temperatures in each mode, and `ntbin` should be set equal to the number of turbulence variables.

The function file should have dimensions of the number of internal cells in each grid block if `idummy` = 0, or the number of internal cells + 2 to account for a single row of dummy cells if `idummy` = 1. The second layer of dummy cells, used for high order flux extrapolations, should never be included in the input function file. Either style can be used to create restart files. If dummy cell information is not provided in the function file, values in the dummy cells will be extrapolated from the interior, and then overwritten by the corrected values when DPLR is run. If the dummy cells are included in the file, the values contained in the dummy cells should be face centered values at all solid surfaces. This allows for an exact specification of the viscous wall boundary condition. If dummy cells are not included, the boundary condition at any viscous walls will be reinitialized on restart, which will lead to a significant perturbation to the flowfield and L2norm residual.

During the conversion process FCONVERT will generate all necessary header elements and format the file properly for DPLR. However, the resulting restart file does not contain all of the CFD modeling flags, and thus cannot be post-processed with POSTFLOW until it has been run at least zero iterations and re-saved in DPLR.

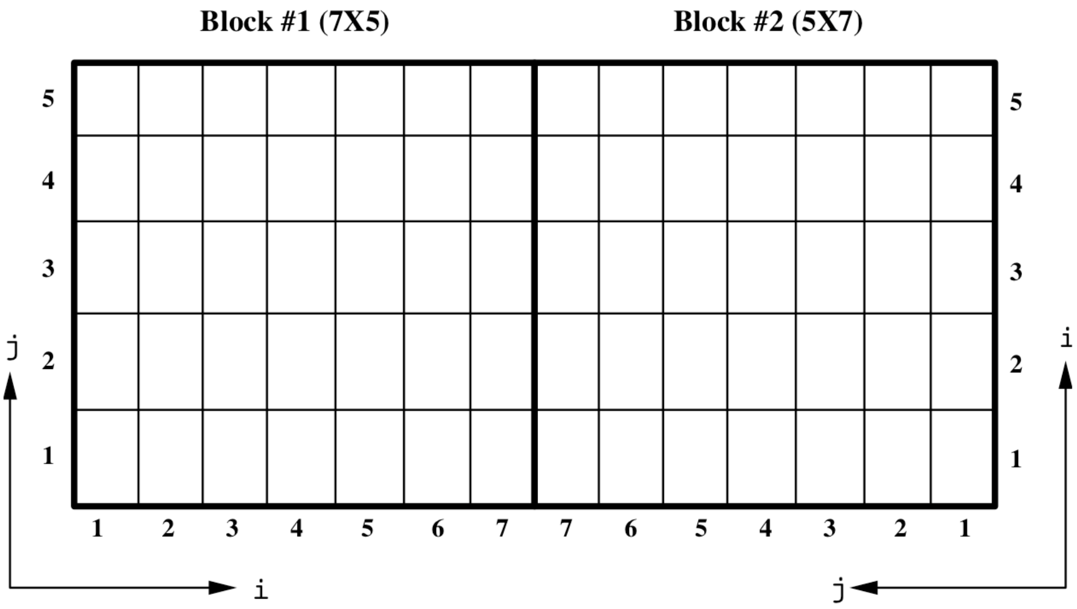

Figure 1. Zonal interface example #1.

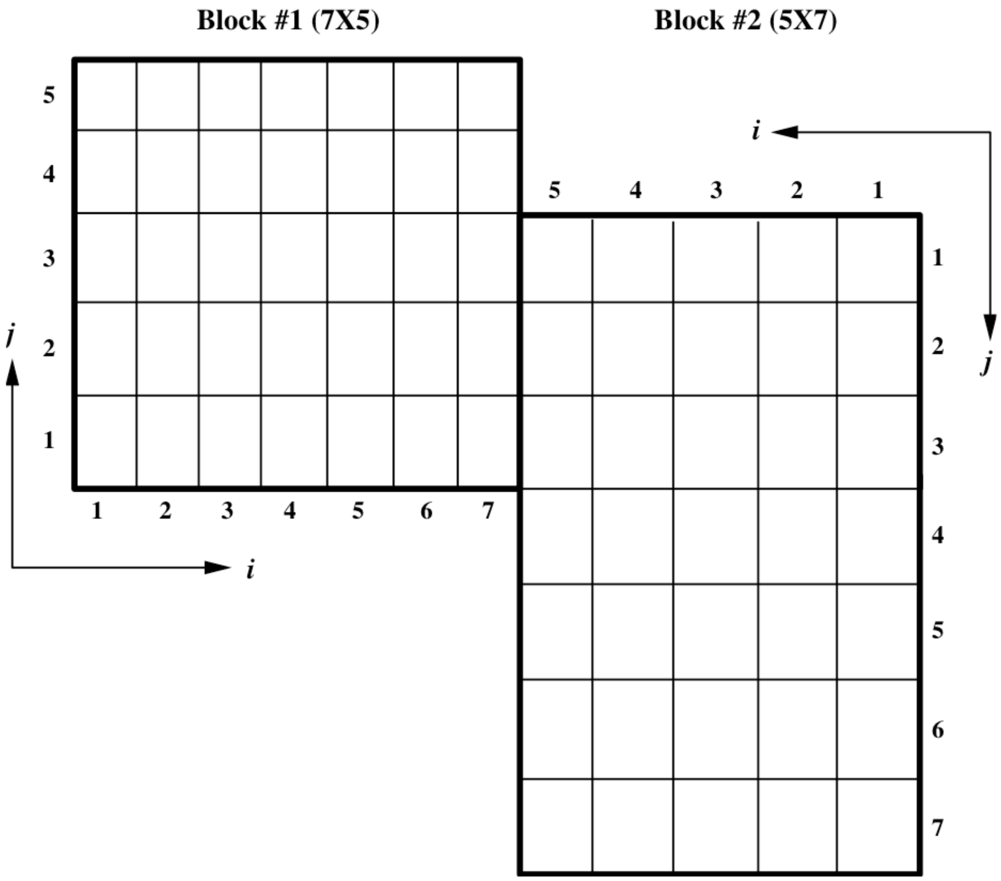

Figure 2. Zonal interface example #2.

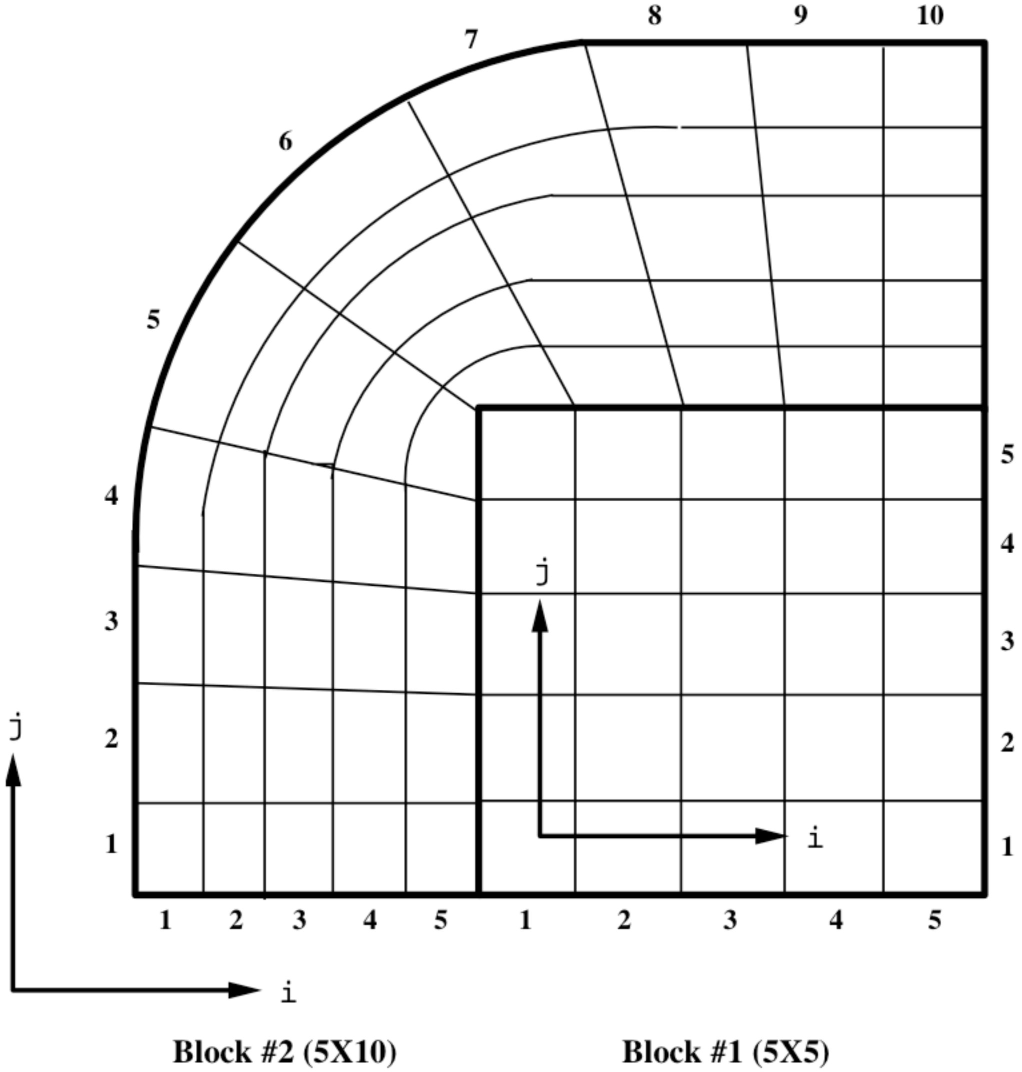

Figure 3. Zonal interface example #3.

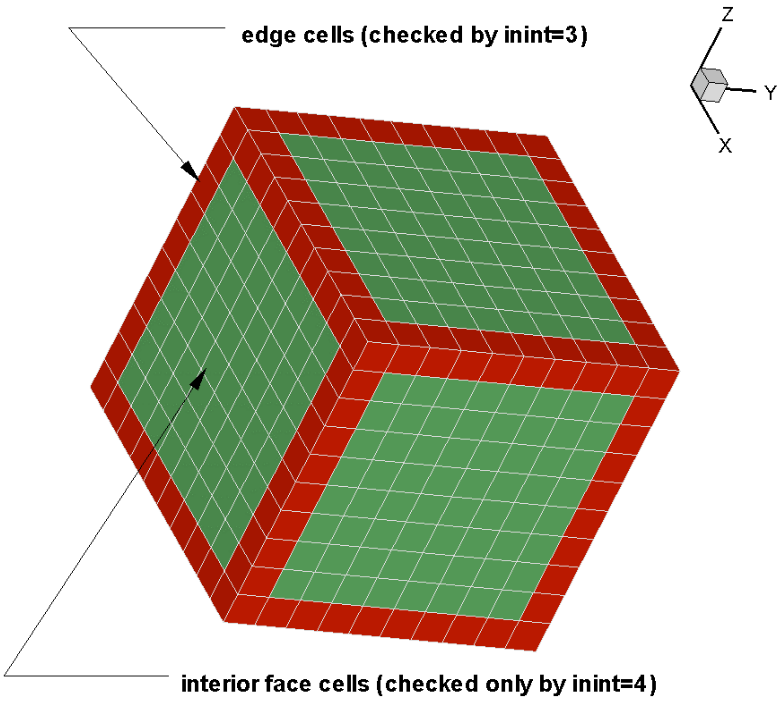
edge cells (checked by inint=3)
Z
Y
X
interior face cells (checked only by inint=4)

Figure 4. Edge cells checked with `inint` = 3 (shown in red).

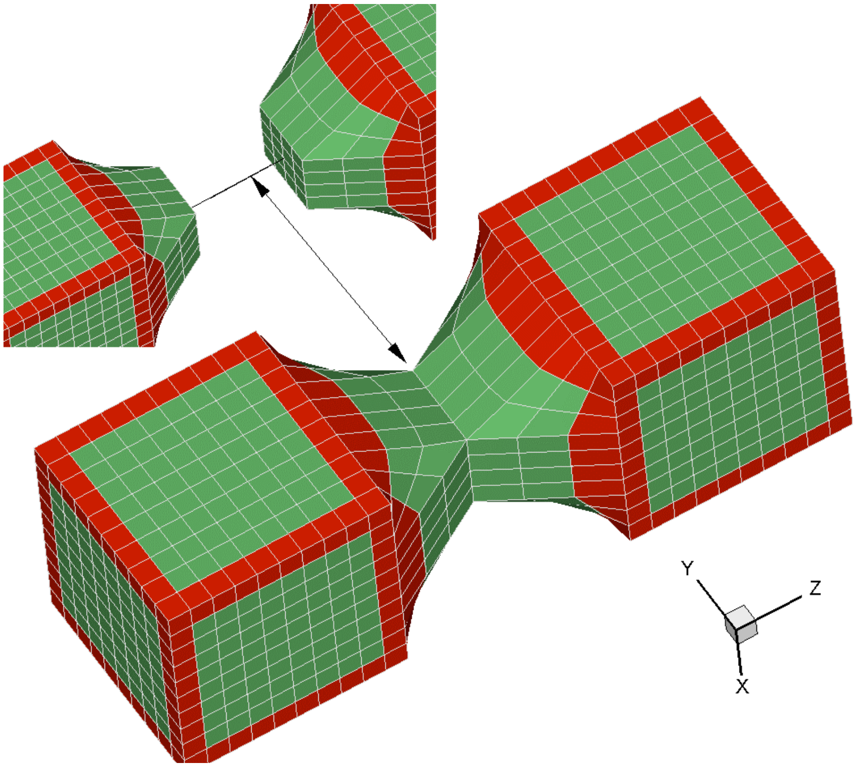

Figure 5. Example of 3D zonal interface detectable only using the `inint` = 4 option.

APPENDIX: Release Notes for Version 3.05.0

UPGRADES:

v3.05.0 -- add imseq=2, all blocks seqenced by same factors

BUGLIST:

v3.05.0 -- FIXED BUG: array bounds bug that did not appear to cause problems anyway (interface)
-- FIXED BUG: issue with some edge interfaces (decompint)

MODLIST:

v3.05.0 -- add imseq=2, all blocks seqenced by same factors (readinp)
-- minor mods for electronic energy stuff (readinp)
-- output formmatting (utils)