NASA Contractor Report 4772

The Volume Grid Manipulator (VGM): A Grid Reusability Tool

Stephen J. Alter
Lockheed Martin Engineering & Sciences Company • Hampton, Virginia

National Aeronautics and Space Administration
Langley Research Center • Hampton, Virginia 23681-0001

Prepared for Langley Research Center
under Contract NAS1-96014

April 1997

Abstract

This document is a manual describing how to use the Volume Grid Manipulation (VGM) software. The code is specifically designed to alter or manipulate existing surface and volume grids to improve grid quality through the reduction of grid line skewness, removal of negative volumes, and adaption of surface and volume grids to flow field gradients. The software uses a command language to perform all manipulations thereby offering the capability of executing multiple manipulations on a single grid during an execution of the code. The command language can be input to the VGM code by a UNIX style redirected file, or interactively while the code is executing.

This code has been written with ANSI FORTRAN 77 and C, offering portability to a wide variety of computer platforms. Since most grid generation codes require Silicon Graphics machines, this code has been thoroughly tested on such machines; though the code has been successfully ported to the Sun architecture. To use this code it is recommended that grid visualization software be used. The Flow Analysis Software Toolkit (FAST) works well, but any good visualizer can be effectively applied. The visualization software is required to view the surface and volume grids resulting from the manipulations performed.

The manual consists of 14 sections. The first is an introduction to grid manipulation; where it is most applicable and where the strengths of such software can be utilized. The next two sections describe the memory management and the manipulation command language. The following 8 sections describe simple and complex manipulations that can be used in conjunction with one another to smooth, adapt, and reuse existing grids for various computations. These are accompanied by a tutorial section that describes how to use the commands and manipulations to solve actual grid generation problems. The last two sections are a command reference guide and trouble shooting sections to aid in the use of the code as well as describe problems associated with generated scripts for manipulation control.

Contents

List of Tables

List of Figures

List of Scripts

Chapter 1

Introduction

Current methods of generating domain discretizations for computational fluid dynamics use the solution of algebraic and partial differential equations (PDE).[1–4] These methods usually offer a variety of options to control grid line incidence at a boundary, cell spacings at a boundary and grid line skewness. These controls usually produce usable grids, but not always. There may be several instances that require an iterative approach to determine the best controls for the algebraic and PDE solvers to obtain a usable surface or volume grid. A few of these instances are:

- conflicting PDE controls for grid-line incidence angle at a boundary;
- improper point spacing at a boundary or in a region;
- poor grid adaptions based on flow field gradients.

Throughout the Computational Fluid Dynamic (CFD) simulation process, there is a need to reduce grid generation time by reuseing existing surface and volume grids. Some of the procedures that can be used to take advantage of existing grids are:

- conversion of inviscid volume grids to viscous grids, and vice versa;
- expansion of grids to guarantee flow field capture;
- changing of vehicle shapes to evaluate favorable aerodynamic characteristic trends (i.e. parametric studies);
- single and multi-faceted grid adaption without the formation of skewed grid lines;
- topological changes to improve flow solver modeling capacity.

The VGM code has been designed to meet all these needs, as well as other grid manipulations. The VGM code is comprised of 11 commands that when combined, provide a powerful tool to manipulate existing surface and volume grids. This manual explains in detail how these commands work and how to use them in conjunction with one another to perform all the manipulations mentioned above and more. This manual has been written to explain the uses of the VGM commands as opposed to describing each command separately.

Each chapter highlights a specific set of manipulations that can be used to augment grid generation, grid adaption, and other grid related issues. Due to the fact that VGM is a language, the commands have multiple uses among the manipulations described; hence the capabilities of each command is spread throughout the manual.

The manual is comprised of 14 sections. The two sections following this introduction describe the memory management and the manipulation command language. The next 8 sections describe simple and complex manipulations that can be used to smooth, adapt, generate, and reuse existing grids for various CFD computations. Each of these manipulations can be used alone or in combination to change an existing grid to fit the users' needs. These sections are accompanied by a tutorial section that explains how to use the commands to solve actual grid generation problems. The last two sections comprise a command reference guide and a trouble shooting section to aid in the use of the code.

Throughout this manual, when commands, command arguments, and VGM structures are described, **bold face** lettering is used to identify actual commands, *italicized* lettering identifies command arguments, {}'s identify optional command arguments, and [ ]'s identify limits of operation for a given array variable or volume grid.

Chapter 2

Memory Management

2.1 Grid Blocks

The VGM command language is built around a single data structure that represents the dimensions of a volume grid. The code assumes that all subsequent grid types, including surfaces, curves, and points, are a subset of a volume grid. The data structure for all grids is: (in command like form)

```
XYZ[ngsys,nblk,I-limits,J-limits,K-limits]
```

where,

ngsys Grid System Number -
This number represents a group of blocks **read** in or created within VGM using the **combine**, or **redist** commands. Every time a new block or set of blocks is generated the *ngsys* maximum value increases and the data is stored in the appropriate array with the *ngsys* denoting the beginning pointer to the data (grid or array variable).

nblk Block Number -
Each grid system has to have at least one block. The *nblk* variable identifies this block and any other block in a specific grid system. For some commands the *nblk* variable can be represented by the "limit" format, discussed below, for identifying a range of blocks.

I-limits First Computational Index Limit

J-limits Second Computational Index Limit

K-limits Third Computational Index Limit

The limits specification is as follows:

```
vb-ve:vc
```

or beginning index (vb) to ending index (ve) by an increment (vc). All computational index limits used in the commands use this construct. Some commands also allow the use of the limits on the block numbers to be operated on. Each of the values can be either a 1 denoting the minimum value, a 0 denoting the maximum value or an actual number between the minimum and maximum limits for a specified grid block or array variable.

The various grid types that are representable occur when one or more of the index limits are single valued. The representation of a surface has one index limit that is single valued, a grid curve has two index limits that are single valued, and a grid point has all three index limits as single valued. There are several rules that must be followed when using the xyz[...] construct. These are:

1. If any limit is missing, the entire range with (vc=1) is assumed.
2. If the beginning or increment value is missing, they are assumed to have the value of 1.
3. If the ending value is missing, it is assumed to be the maximum value possible for a specified grid block or array variable.
4. If only one number is available for the entire range, the minimum and maximum value is that number, with the increment set to 1.
5. If the increment value (vc) is 0, the actual value is the maximum value subtracted from the minimum value.

For example:

```
xyz[1,1,1-0,1-5,3-22:4]
```

results in a volume being defined with the following characteristics:

$ngsys = 1$
$nblk = 1$
I-limits = I_{min} to I_{max} by 1
J-limits = J_{min} to 5 by 1
K-limits = 3 to 22 by 4

and,

```
xyz[5,4,3,0,10]
```

results in a point being defined as:

$ngsys = 5$
$nblk = 4$
I-limits = 3 only
J-limits = J_{max} only
K-limits = 10 only

2.2 Internal Variables

The VGM code also has internal variables that can be used for input to various commands. There are two types, *core* or array, and *temporary* constant variables. The *core* variables are generated with an **allocate** command and are referenced similarly to the grid blocks. For example:

```
dsj1[1-0,,1-91:0]
```

results in the following values being assigned:

core variable = dsj1 (which may be a computed arclength in the J-direction
I-limits = I_{min} to I_{max} by 1
J-limits = J_{min} to J_{max} by 1
K-limits = K_{min} to 91 by 90

Constant variables (i.e. *temporary*) are single valued, as compared to the *core* variables.

2.3 Memory Limits

The VGM command language has several limits with respect to command arguments and, grid size and array space. These limits include:

Number of Grid Systems	=	100
Number of Grid Blocks	=	200
Number of Grid Points	=	8,000,000
Number of Array Variables	=	40
Number of Array Data Points	=	8,000,000
Number of Constant Variables	=	40
Number of Command Arguments	=	40
Number of Consecutive Command Lines	=	1,000

Table 2.1: VGM Memory limits; both computer and language.

Violation of these limits usually results in an error. If an error occurs, the code will stop execution in a batch mode and stop processing a command in the interactive mode.

Chapter 3

Language Specifics

As with any computer language there are specific characteristics to the command structures and how they can be used together to operate on data. The VGM code is no different. Special attention has been paid to ensure this language is as consistent as possible and as general as possible without sacrificing flexibility. Also, each of the arguments are descriptions of the flags they are setting, thereby making an input deck or script easy to read. Both the command key words and the arguments do not need to be capitalized, as they will be converted to the lower case.

3.1 Command Argument Ordering

Each command in the VGM code starts with a key word. This key word tells VGM which command is to be issued. The commands and their uses are:

Command	Description
allocate	Create a new block or array variable
blend	Interpolate between existing points in a variable
combine	Regroup volume grids into a single grid system
copydist	Copy a distribution from one grid line to another
quit	End execution
read	Input data
redist	Redistribute a grid line based on a function
set	Equate variables and data or compute data
smooth	Smooth a grid with algebraic or PDE solvers
tfi	Perform Trans-Finite Interpolation on a region/zone
write	Output data

Table 3.1: VGM command summary.

After each command there are any number of arguments. The order of the arguments is *not* fixed. As long as all the arguments are available in the command line, VGM only requires that the arguments come after the key word. There are two exceptions. The **copydist** command requires a source and destination grid for the copying of grid-line distributions from one grid to another. The order of these two grids is important, as explained in chapter 7. The **combine** command requires a list of blocks to be combined to for a multiple block data set, where the order of these block is based on the order of the block identifiers. In addition, the **set** command has a source and destination for data which is order dependent, as explained in chapter 9. The **copydist combine** and **set** commands are the *only* commands that require order specific arguments.

3.2 Script Readability

The VGM scripts or input decks have several features that improve readability. First, as mentioned above, the commands and the arguments need not be capitalized because they will be converted to the lower case. File names containing a mixture of upper and lower case letters will be read as specified. Internally generated variables will be converted to lowercase, so uppercase characters in variables will have no effect.

Second, the language allows for the continuation of a command and its arguments onto multiple consecutive lines by appending a \ to the end of each line. Beware, the number of arguments in a command line are limited. The limits are discussed in chapter 2 section 2.3. VGM does not allow partial arguments to be on separate lines.

Third, the language allows blank lines and indentation of command lines. Whether the command is entered in batch or interactively, the blank lines have no effect on the final grid manipulations. However, if more than 9 consecutive blank lines are in a file, the code will stop, because it may have encountered a file with no ending command.

Fourth, the language has comment lines to explain the manipulations about to be done. It is *strongly recommended* that the comment lines be used in batch operated scripts because the commands can be used over and over again in a series, resulting in serious confusion on the manipulations being performed. The comment line is marked by placing a # as the first character of the line. The # can not be indented, but the following text can be spaced at the users' discretion.

All of these attributes are designed to make the scripting language more readable and understandable. They are not intended to confuse the reader of the scripts, they are only available to make the script "legible". Thus, use of the last two capabilities is recommended for easy explanation of manipulations to be done on a surface or volume grid.

3.3 Script Progress and Results

Due to the nature of this code and the language, results of a manipulation is output to the user. Each command will generate a set of information that should be used to identify grid system numbers, block numbers and limits of any manipulation. It is *recommended* that the user read the output and save it, if possible, for trouble shooting. The output from each command is printed to the standard output device in UNIX, and is augmented by the debugging file produced from each execution of the VGM code as explained in chapter 14.

Chapter 4

Input and Output

4.1 Commands

In order to manipulate existing grid data, a method of inputting and outputting grids is needed. The VGM code makes use of two simple commands **read** and **write** to input and output data from various computer software, including LAURA,[5] GRIDGEN,[2] Tecplot,[6] and PLOT3D.[7] The VGM command syntax for **read** and **write** is as follows:

read *filename* {*type*} {*style*} {*format*} {*dimension*}

where,

filename is the file name of the data to be **read**. The file name rules are as follows:

1. Limited to 60 characters in length;
2. Can not be identical to **read** arguments;
3. Can not contain ['s,]'s, \'s, or commas;
4. Are case sensitive;
5. May contain directory placement characters (./, ../ and ~)

format is the data format, `ASCII`, `unformatted`, or `binary`. <default=`unformatted`>

style is the style the file is in; `gridgen`, `plot3d`, `laura`, or `tecplot`TM. <default=`plot3d`>

type is the type of data in the file; `gridonly`, `solution(`*ngsys*`)`, or `curve`. <default=`gridonly`>

NOTE: The solution(*ngsys*) option requires a grid system number to attach the data to, to ensure there is one value for each grid point in each block. The variables loaded in this manner will have variable names of the form:

varname`_nNN_blkBBB`

where the NN represents the Grid System number and the BBB represents the block number.

dimension represents the number of blocks in the grid set, `single` or `multiple`.
<default=`single`>

```
write filename {type} {style} {format} {dimension} xyz[...]  {orientation}
```

where,

filename is the file name of the data to be **written**. The file name rules are identical to the **read**command.

format is the data format, `ascii`, `unformatted`, or `binary`. <default=`unformatted`>

style is the style the file is in; `gridgen`, `plot3d`, `laura` or `tecplot`(*variables*).
<default=`plot3d`>

NOTE: The variables specifiable in the `tecplot`(*variables*) option include the physical coordinates (X, Y, and Z), the computational coordinates (I, J, and K), and *array* and *constant* variables in the form:

```
tecplot(x,y,z,i,j,k,dsj1)
```

type is the type of data in the file; `gridonly`, `solution`(*ngsys*), or `curve`.
<default=`gridonly`>

NOTE: The solution(*variables*) option requires a set of variables, similar to the `tecplot(...)` argument.

dimension represents the number of blocks in the grid set, `single` or `multiple`.
<default=`single`>

xyz[...] is the block or region or set of blocks to be written as a data set.

NOTE: The block limits may be used in this command to select a range.

orientation is the physical and computational orientation of the grid. It is specified with the following argument:

```
switch(x,y,z,i,j,k)
```

where the physical coordinates are specified in the order to be written, and same with the computational coordinates.

NOTE: The *orientation* basically changes the entire reference frame of the grid written. Beware, no check is done to determine if a left handed coordinate system is written.

As stated in the introduction, the arguments in the { }'s are optional. The default values for these optional arguments are given, but the **write** command does not use code initialized defaults for grid systems that were read. The input settings used to **read** in the grid systems is used as the defaults for the same grid systems when written. These can be overridden by specifying the arguments on the command line.

4.2. READ and WRITE Usage

4.2.1 PLOT3D

The following is an example illustrating the use of the **read** and **write** commands: These

```
read shuttle.g plot3d single unformatted grid
write tail.vol single unformatted grid plot3d xyz[1,1,121-0,1-15]
```

Script 4.1: External grid extraction.

commands **read** a grid from the file named "shuttle.g" using the PLOT3D style, single block dimension, unformatted format, and **write** a grid volume encompassed by I=121 to I_{max}, J=1 to 15 and all K points. By using the defaults, the VGM script 4.1 can be reduced to:

```
read shuttle.g
write tail.vol [1,1,121-0,1-15]
```

because the file is a PLOT3D style, unformatted format, single-block dimension grid. As stated in the previous chapter, there is *no* order to the arguments of these commands.

If the file was multiple block and/or in ascii, the defaults would no longer hold on the **read** command. But if the output file was to be identical, the defaults on the **write** command would be those set by the **read** command. For example:

```
read shuttle-MULTI.g multiple ascii
write tail.vol [1,3]
```

would result in `tail.vol` being a PLOT3D style, ascii format, multiple-block dimension grid, only containing one block. The PLOT3D style is the default on the **read** so it would also be default on the **write** in this case only.

4.2.2 GRIDGEN

As most of the PLOT3D formats are supported, so to are the GRIDGEN surface and curve formats. To load in a *.mlga file that contains 6 surfaces per grid block, use the following:

```
read shuttle.mlga multiple ascii gridgen
```

The surfaces are then loaded as a series of 6 surfaces, each with I and J indices. The K-index faces are not recognized because the style only has two index maxima. Likewise, to load in a single surface (*.grda) just omit the `multiple` argument because the `single` block grid dimension is the default.

To load in a GRIDGEN curve, use:

```
read edge1.dat ascii gridgen curve
```

Again, since there is only one computational index, the curve is loaded as an I-varying grid line.

Similarly on output, the following can be used to output a *.mlga file and a curve, respectively: The curve that is written in this case is from grid system number 1, block 5,

```
write shuttle.mlga multiple ascii gridgen xyz[1,1-6]

write edge3.dat ascii gridgen xyz[1,5,6,15,2-57]
```

Script 4.2: Gridgen face and curve extraction.

with I=6, J=15, and K varying from 2 to 57. VGM will look for the index that is varying and only output that curve. The `curve` argument is not needed on the output because the `xyz[...]` has only one varying index, and the style is GRIDGEN.

4.2.3 Solution

If a volume grid has already been read into VGM, a solution can be read from various types of input and attached to the grid points. This is done by using the *solution*(*ngsys*) option of the **read** command:

```
read shuttle.q solution(1) plot3d
write tail.q solution(1) xyz[1,3]
```

Script 4.3: Solution I/O manipulation.

instructs VGM to read in the solution file `shuttle.q` for an existing volume grid, and attach it to grid system number 1. This file is expected to be the standard PLOT3D solution style for a "Q"file. When the data is referenced on the output, the grid system number is 1 in the `solution(1)` argument and block 3 of grid system 1 is to be written. Although the 1 in the `solution(1)` argument is redundant, it is specified for consistency.

There are other ways to write the solution for a grid, besides using the grid system number in the `solution(1)` argument of the **write** command. The `solution()` argument may also contain variable names. For example:

```
write tail.q solution(rho,u,v,w,e) xyz[1,3]
```

Script 4.4: Variable specific solution output for PLOT3D style.

is acceptable providing that the variables `rho_n1_b3`, `u_n1_b3`, `v_n1_b3`, `w_n1_b3`, and `e_n1_b3` are defined and have the same I-, J-, and K-dimensions as the `xyz[1,3]` grid block. Assuming the PLOT3D style, the data will be written as a PLOT3D solution file. If too few or too many variables are selected in the `solution()` argument, the code will not write the file and an error will result.

4.2.4 TECPLOT™

Another way to **read** and **write** data is using the `tecplot(variable1,variable2,...)` argument. The `tecplot()` argument allows for the input of TECPLOT™ ascii data sets and the writing of data in the TECPLOT™ block format. For example:

```
read shuttle.dat tecplot ascii
write tail.q ascii tecplot(rho_n1_b1,u_n1_b1,v_n1_b1,w_n1_b1,e_n1_b1) xyz[1,3]
```

Script 4.5: Variable specific solution output for TECPLOT style.

will read a TECPLOT™ file and assign the physical coordinates to an xyz[...] grid block, and the other variables based on the `solution(ngsys)` algorithm described above. On output, the `tecplot()` command must contain those variables to be written. The physical coordinates X, Y, and Z, and the computational coordinates I, J, and K are plain letters in the variable list. Other variables have to be named exactly. In this case, since the data was read in as a TECPLOT™ data set, the density (`rho`), the three velocity vectors (`u, v, and w`) and the energy (`e`) are specified with their respective grid system and block numbers used in the reading process to generate the variable names.

> **NOTE:** The variables listed in the output are *not* used in the input because they already reside in the file. Also, on output, all variables and coordinates have to have the same dimension or the data can not be written.

4.2.5 Re-orienting Coordinates

The `switch(x,y,z,i,j,k)` argument is a powerful tool to change the orientation of a grid. As explained above, the `switch()` argument allows for the re-orientation of a volume grid by changing the physical and computational coordinates through the writing of them in succession. VGM assumes that X, Y, and Z are written in I, J, and K computational coordinates, but the orientation can be changed to anything, provided that all 6 coordinates are written and not repeated. If any of the physical coordinates has a "-" in front, the sign of the coordinate is reversed. Likewise, if the computational coordinate has a "-" in front, that index will be written in reverse order. For example:

```
switch(y,-z,x,k,-i,j)
```

instructs VGM to write the Y-coordinate first, then the reversed Z-coordinate, then the X-coordinate. The computational order of the data will be the current K-index first, then the I-index in reverse order, and finally the J-index.

The `switch` argument can be used with any of the styles, as the orientation only affects the reference frame of the grid data.

4.3 Manipulative Capabilities

The **read** command has no manipulative capability. It is specifically designed to input data. But the write command has many manipulative capabilities. These include:

1. Converting grid data from one style and format to another.
2. Coarsening an existing volume grid for grid convergence studies.
3. Extracting surface and curve grid data from volume grids.
4. Re-orienting a surface grid from a GRIDGEN *.mlga file to use in the generation of a volume grid.
5. Evaluating grid data by printing selected quantities.

4.3.1 Converting Grid Styles

In dealing with multiple grid generation codes such as GRIDGEN,[2] 3DGRAPE/AL[8] and 3DMAGGS;[1] CFD codes such as LAURA,[5] GASP[9] and TLNS3D;[10] and visualization tools such as FAST[11] and TECPLOT^{TM},[6] each require a surface or volume grid. The style used by each may be different, but the data is all the same. The VGM code can accommodate most codes requiring surface and volume grids through the PLOT3D, GRIDGEN, TECPLOT and LAURA grid data styles. Reading with one style and writing with another is trivial. The only complexity that arises is when using surface and curve grids in volume grid styles and vice versa.

When GRIDGEN styled data file is read, it only has I and J maxima. To get the K-dimension, the VGM code assumes it to be 1. This is significant when writing data to be used in PLOT3D style because the GRIDGEN surface may indeed have a K-dimension but VGM does not know this. To ensure the K-dimension is written, the `switch()` command can be used:

```
read shuttle.mlga gridgen multiple ascii
write face3.g plot3d single binary xyz[1,5] switch(x,y,z,j,k,i)
```

Script 4.6: Coordinate transformation manipulation.

The above command reads a *.mlga file containing at least 6 surfaces. Using the TEAM[12] nomenclature, face 3 is a constant J face, and when written in the PLOT3D style, the face will have an I and K maxima while the J maxima will be one because the "k" in the `switch(x,y,z,j,k,i)` command is 1.

When converting to and from the TECPLOT^{TM} style, the grid dimension (single or multiple) is not specified. Rather it is located in the data file. If this type of file is read, the user must beware of the dimensionality of the block so as to prevent confusion on the computational coordinate limits.

4.3.2 Coarsening Grids

In the evolution of CFD simulations the necessity to do grid convergence studies usually arises. Or to start a CFD simulation, it may be simpler to establish a flow field about a configuration using a coarse mesh of the the discretized domain. In either case, the volume grid can be coarsened in any of the computational coordinate directions by simply using a non unity increment for the grid block to be written. For example:

```
read shuttle.g
write shuttle_c8.g xyz[1,1,1-0:8,1-0:8,1-0:8]
```

will read a volume grid from the `shuttle.g` file in PLOT3D style and write out a volume grid skipping 8 points in all three computational directions. If the grid is "multigridable" by 3 levels, 1 subtracted from the computational dimension limits will be divisible by 8. If this is not the case, the last point in the direction that is not divisible by 8 will *not* be written. For example, if the computational limits are (161 X 133 X 65), with the skipping of every 8 points, the new dimension will be (21 X 17 X 9) because:

$$I_{\text{dim}} = \frac{161-1}{8} + 1 = 21.0 = 21 \text{ points}$$

$$J_{\text{dim}} = \frac{133-1}{8} + 1 = 17.5 = 17 \text{ points}$$

$$K_{\text{dim}} = \frac{65-1}{8} + 1 = 9.0 = 9 \text{ points}$$

In actuality, the limits should be (21 X 18 X 9) to get the last point. With the full dimensions, the 17 point would correspond to the J=129 which is 4 points shy of the limit at 133. But the manipulation is a simple one and can be used over and over again for successive coarsening.

4.3.3 Surface and Curve Extraction

In the improvement an existing grid it becomes necessary to change defining grid block surfaces. To change these surfaces, VGM can be used, but other tools such as GRIDGEN and GRIDTOOL[13] are viable for capabilities not in VGM, such as elliptic solving and surface projection, respectively. To improve or change a surface in a volume grid, the VGM code can be used to extract the surface by using the `xyz[...]` argument in conjunction with the GRIDGEN style. For example:

```
read shuttle.g
write face4.grdb gridgen binary xyz[1,1,,0] switch(x,y,z,k,j,i)
```

will read in a volume grid representing the shuttle and write out a surface in GRIDGEN style in K,I varying indices. Though these indices can be I,J or any other pair, in the TEAM nomenclature, face 4 is a J-constant face with K,I varying indices. The VGM code will detect the switch in coordinates and write out a surface grid for further manipulation, because the 0 in the `xyz[1,1,,0]` specifies the maximum J index and the missing I and K ranges specify the entire I and K index limits.

4.3.4 Re-orienting Grid Data to Generate Volume Grids

The GRIDGEN3D, 3DGRAPE/AL, and 3DMAGGS codes can all generate volume grids based on the algebraic solution of Trans-Finite Interpolation (TFI). VGM can do the same, but only requires the *.mlga file from GRIDGEN or GRIDGEN2D. To generate the volume grid, the surfaces in the *.mlga file need to be oriented in the computational domain to be properly placed in the volume grid block definition. To do this, the `switch(x,y,z,i,j,k)` command is used to take the assumed I-J varying faces into I-J, J-K, and I-K faces using the following commands:

```
#
# Extract individual faces from a GridGen *.mlga file and re-orient
# them to fit within a PLOT3D grid block:
#
read shuttle.mlga gridgen ascii multiple
write face1.g plot3d binary single xyz[1,3] switch(x,y,z,k,i,j)
write face2.g plot3d binary single xyz[1,3] switch(x,y,z,k,i,j)
write face3.g plot3d binary single xyz[1,3] switch(x,y,z,j,k,i)
write face4.g plot3d binary single xyz[1,3] switch(x,y,z,j,k,i)
write face5.g plot3d binary single xyz[1,3]
write face6.g plot3d binary single xyz[1,3]
#
# Read in the new faces:
#
read face1.g
read face2.g
read face3.g
read face4.g
read face5.g
read face6.g
```

Script 4.7: GRIDGEN faces to PLOT3D block conversion.

After executing this sequence, the VGM code will have six new surfaces ready for assigning to a grid block definition:

> `xyz[2]`, `xyz[3]` are constant I faces;
> `xyz[4]`, `xyz[5]` are constant J faces;
> `xyz[6]`, `xyz[7]` are constant K faces.

Using the **set** command as explained in chapter 11, the volume grid can easily be built with VGM via 3DTFI.

Chapter 5

Grid Based Parameters for Manipulations

One of the most powerful capabilities of the VGM code is the computation of grid related parameters. Grid related parameters can be used to perform various manipulations, including the generation of distribution functions upon which smoothing can be done or transition from one grid to another in a blending type operation. These internally computed parameters form the basis for most of the complex grid manipulations that VGM is capable of doing.

The grid related parameters are primarily arclength and normalized arclength parameters, based on a computational direction of a surface or volume grid. The six possible grid parameters or intrinsics are listed in table 5.1:

`dsia` =	Arclength function in I-direction
`dsja` =	Arclength function in J-direction
`dska` =	Arclength function in K-direction
`dsin` =	Normalized arclength function in I-direction
`dsjn` =	Normalized arclength function in J-direction
`dskn` =	Normalized arclength function in K-direction

Table 5.1: VGM grid parameters (intrinsics).

The difference between the normalized arclength and the standard arclength is the normalized arclength varies from 0.0 to 1.0 in the computed direction (see fig. 5.1). The normalized arclengths are typically used to redistribute a grid within its physical limits, and the standard arclengths are used to change the physical limits of a grid.

These parameters are based on surface and volume grids through the following syntax:

```
dska(xyz[...])
```

where the `xyz[...]` is the region of a grid block to be used.

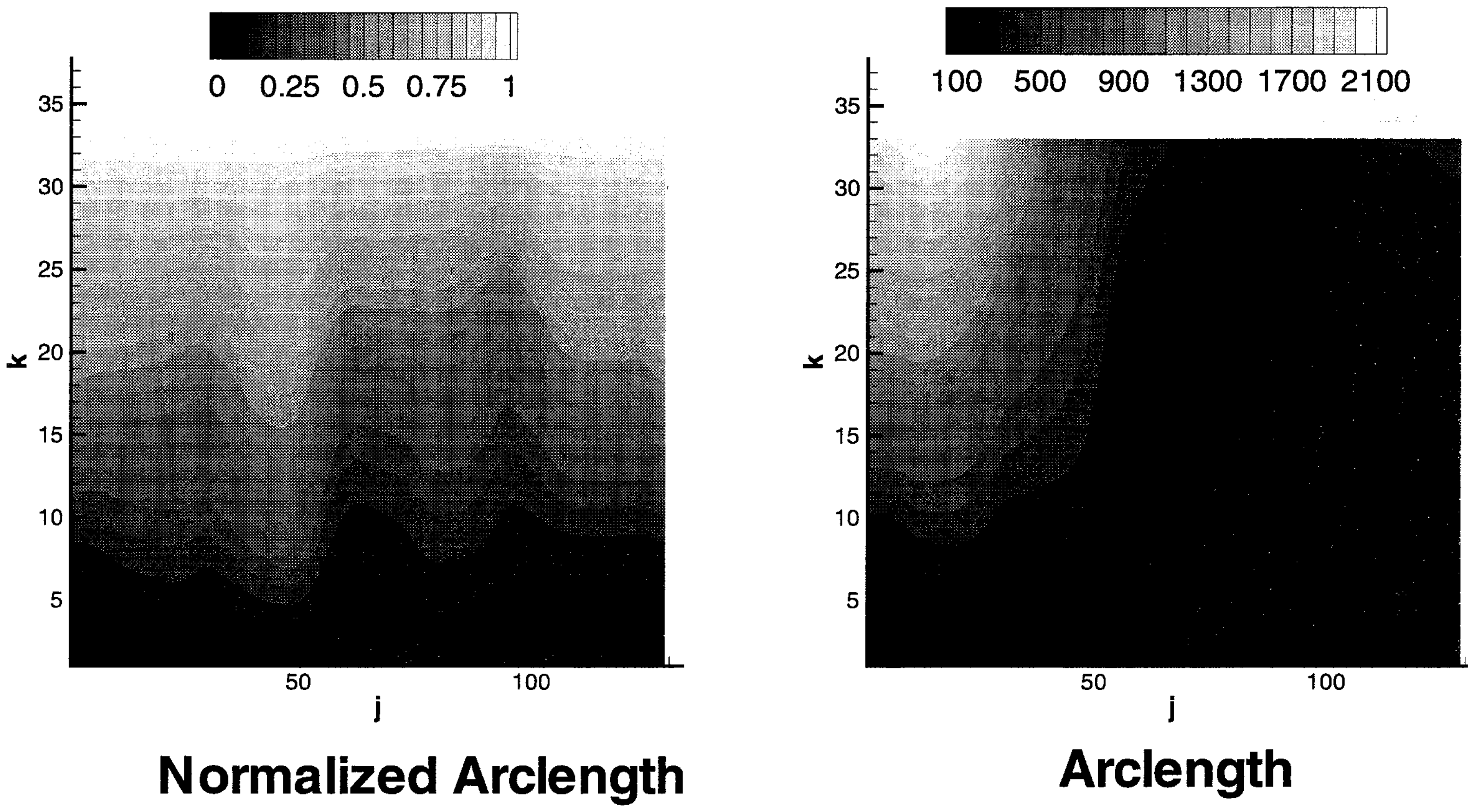

Figure 5.1: Arclength parameter space differences.

5.1 Computing Grid Parameters

Prior to computing these grid parameters, a variable array has to be created to store them for later use. To create a variable array the **allocate** command is used:

$$\boxed{\textbf{allocate}\ \ \textit{varname[I-limit,J-limit,K-limit]}}$$

where,

varname is the array variable name to store computed grid parameters. The array variable name rules are as follows:

1. Limited to 60 characters in length;
2. Can not be identical to grid parameters (intrinsics);
3. Can not contain ['s,]'s, \'s, or commas;
4. Are not case sensitive;
5. Computational limits may not exceed the grid-point limits of VGM.

I-limits First Computational Index Limit

J-limits Second Computational Index Limit

K-limits Third Computational Index Limit

The **allocate** command reserves memory for a computed intrinsic. It can also be used to allocate a new grid block (see chapter 6). Once the variable has been created, the **set** command is used to compute and assign the results of the computation to the variable:

```
set varname1[I-limit,J-limit,K-limit] = varname2[I-limit,J-limit,K-limit]
```

-or-

```
set varname1[I-limit,J-limit,K-limit] = ds*(xyz[ngsys,nblk,I-limit,J-limit,K-limit])
```

where,

varname1 is the destination array variable name to store computed grid parameters or other variables.

varname2 is the source array variable or intrinsic (ds*) to be equated or computed, respectively. The only rule that must be followed is the computational region of each variable or intrinsic in the equate must be the same.

I-limits First Computational Index Limit of region to be **set**

J-limits Second Computational Index Limit of region to be **set**

K-limits Third Computational Index Limit of region to be **set**

NOTE: The **set** command has many other capabilities that will be discussed later.

The capabilities of the **set** command also include:

- Extract grid data (see chapter/section 6.1);
- Insert grid data (see chapter/section 6.1);
- Merge grid data (see chapter/section 6.2); and
- Shift grid data (see chapter/section 11.2);

Once computed, these arclength parameters can be used as input into various VGM commands for controlling cell sizing, grid-point distribution, and grid smoothness. All of these functions are described in more detail in chapters 8 and 9. For grids that are to remain within the same physical limits, the normalized arclength is the best choice as it can be used to change a grid distribution without changing the length of the grid-line being altered. Standard arclengths should be used for grids that are to be changed in both distribution and grid-line length.

5.2 SET and ALLOCATE Usage

With the **allocate** and **set** commands, various grid parameters can be computed. For example, to compute the distribution function in the I-direction of a volume grid with every other point in the J-direction, use the following:

```
allocate dsi1[161,33,33]
set dsi1 = dsia(xyz[2,4,11-171,57-121:2,1-33])
```

Script 5.1: Computation of I-direction based arclength parameter.

Or, to compute the normalized arclengths in separate regions of a volume grid, use:

```
allocate dsi2[129,197,65]
set dsi2[,1-17] = dsin(xyz[1,1,,1-17])
set dsi2[,41-73] = dsin(xyz[1,1,,57-89])
set dsi2[,101-121] = dsin(xyz[1,1,,141-161])
```

Script 5.2: Computation of I-direction based normalized arclength parameter.

The last example may be the matching of interfaces from one grid to another. At these interfaces, the grid may require smoothing or improved interfacing, so the normalized arclength function may be manipulated (see chapter 9). Regardless of the reason, grid parameters can be computed by VGM for further processing.

Some of the inputs that can be generated from the **allocate** and **set** commands for grid manipulations are cell heights. To compute cell heights, the following can be used to generate a surface variable containing cell sizes to be used for re-distributing:

```
allocate dsi2[129,197,2]
allocate dsi2a[129,197,1]
allocate dsi2b[129,197,1]

set dsi2 = dsia(xyz[1,1,,,1-3:2])
set dsi2a = dsi2[,,2]

set dsi2 = dsia(xyz[1,1,,,32-33])
set dsi2b = dsi2[,,2]
```

Script 5.3: Endpoint cell sizes for a region in the I-direction.

These computations store the distance from the K=1 boundary to the K=3 boundary as the cell heights in variable `dsi2a` and the *current* cell heights at K_{max} in variable `dsi2b`. This information can be very valuable for Vinokur's[14] function used in a re-distribution (see chapter 8).

Chapter 6

Extracting, Inserting, Merging and Combining Grids and Grid Parameters

One of the most important aspects of grid generation is topology. The topology determines what a grid will look like as well as how well it can model a flow field about a complex aerodynamic configuration. In the evolution of a volume grid, it usually becomes necessary to consider blocking strategies that may or may not fit within the chosen topology. These different blocking strategies may be used to improve parallel processing load balancing, improving grid smoothness across grid block boundaries, and even change the overall topology. Modification of existing volume grids to reflect changes in blocking strategies can be difficult and time consuming.

However, using the VGM code to change blocking strategies significantly reduces the complexity by using the **allocate** and **set** commands repetitively. The syntax for these forms of the **allocate** and **set** commands are:

allocate *xyz[I-limit,J-limit,K-limit]*

where,

xyz is a volume grid block, using the standard data structure.

I-limits First Computational Index Limit

J-limits Second Computational Index Limit

K-limits Third Computational Index Limit

NOTE: The grid system number and block number are not included in the **allocat**ion of the new grid block; only the computational limits are required. Also, this command will cause the grid system maximum to increase by 1 each time it is used.

and,

set *xyz[ngsys,nblk,I-limit,J-limit,K-limit] = xyz[ngsys,nblk,I-limit,J-limit,K-limit]*

where,

xyz on the left hand side is the destination block to store the results from extracting or merging grid blocks.

xyz on the right hand side is the source block to be extracted or merged. The only rule that must be followed is the computational region of each grid block in the equate must be the same.

I-limits First Computational Index Limit of region to be **set**

J-limits Second Computational Index Limit of region to be **set**

K-limits Third Computational Index Limit of region to be **set**

NOTE: The **set** command has many other capabilities that are discussed in chapters 5 and 11.

The extraction, insertion and merging of grid blocks are easily accomplished with these commands. But in some cases, it may be necessary to regroup a series of blocks into a single grid system. This can be done using the **combine** command:

combine *xyz[ngsys1,nblk1] xyz[ngsys2,nblk2] ...*

where,

xyz is a source grid block. Subsequent *xyz*'s are other blocks to be added. There are some rules that can be used to govern which grid blocks are used:

1. `xyz[ngsys]` will get all the blocks in grid system `ngsys`
2. `xyz[ngsys,nblk_begin-nblk_end:nblk_increment]` will get those blocks that are referenced in the range from nblk_begin to nblk_end by nblk_increment

NOTE: This command will cause the increasing of the grid system maximum by 1 each time it is used.

When using the **combine** command, the computational limits are not used. If a portion of a grid block is to be used in the regrouping, an extraction should be done prior to the **combine**. The order of the blocks is significant in this command as the order determines which is the first block to the last. Also, the maximum number of arguments on a command line is 40, so if the number of arguments needed to generate a multiple block decomposition is larger than 40, multiple **combine** commands can be used to build up the final set of grid blocks.

6.1 Extracting and Inserting

To extract a grid block, the following is used:

```
allocate xyz[11,17,65]
set xyz[2] = xyz[1,4,21-31,1-17]
```

Script 6.1: Internal grid extraction.

The grid block dimensions of the source would have to be at least (31 X 17 X 65) because these are the maximum limits in the right hand side `xyz[...]` term.

To insert a grid block, the following is used:

```
set xyz[1,4,21-31,17-35] = xyz[2]
```

The limits are similar to the extraction, but these commands in succession would produce a copy of the exact same grid in the exact same physical location but different computational locations.

6.2 Merging Grid Blocks

As stated above, some times it is necessary to merge multiple blocks into a single block for topological changes. To merge multiple blocks into a single block, a single grid block with large enough dimensions needs to be created to accommodate the source blocks. For example, the following is a listing blocks to be combined for a NASA proposed SSTO:[15]

Block Number	Computational Limits	Block Number	Computational Limits
1	(81 X 65 X 33)	7	(41 X 21 X 33)
2	(81 X 65 X 33)	8	(41 X 23 X 33)
3	(41 X 21 X 33)	9	(41 X 23 X 33)
4	(41 X 45 X 33)	10	(41 X 30 X 33)
5	(41 X 57 X 33)	11	(41 X 28 X 33)
6	(41 X 9 X 33)	12	(41 X 9 X 33)

Table 6.1: Initial blocking strategy for NASA proposed SSTO.

These blocks are arranged, both computationally and physically in Fig. 6.1. To generate a single block volume grid from this set of blocks, the dimensions in each series of blocks in each computational direction need to be summed. The series in the J-direction are blocks 1-2, 3-6, and 7-12; and in the I-direction the blocks are the same sets or 1, 3, and 7. Since the K-direction is constant at 33 points, the computational limits are computed by:

$$
\begin{aligned}
I_{\text{max}} &= (81 + 41 - 1) + 41 - 1 = 161 \\
J_{\text{max}} &= 65 + 65 - 1 \\
&= ((21 + 45 - 1) + 57 - 1) + 9 - 1 \\
&= ((((21 + 23 - 1) + 23 - 1) + 30 - 1) + 28 - 1)\ 9 - 1 = 129 \\
K_{\text{max}} &= 33
\end{aligned}
$$

To generate the single block volume grid the following commands are used:

```
allocate xyz[161,129,33]
set xyz[2,1,1-81,1-65] = xyz[1,1]
set xyz[2,1,1-81,65-0] = xyz[1,2]
set xyz[2,1,81-121,1-21] = xyz[1,3]
set xyz[2,1,81-121,21-65] = xyz[1,4]
set xyz[2,1,81-121,65-121] = xyz[1,5]
set xyz[2,1,81-121,121-0] = xyz[1,6]
set xyz[2,1,121-0,1-21] = xyz[1,7]
set xyz[2,1,121-0,21-43] = xyz[1,8]
set xyz[2,1,121-0,43-65] = xyz[1,9]
set xyz[2,1,121-0,65-94] = xyz[1,10]
set xyz[2,1,121-0,94-121] = xyz[1,11]
set xyz[2,1,121-0,121-0] = xyz[1,12]
```

Script 6.2: Grid block merging.

Now that the volume grid is a single block, the grid can be modified as needed. If the blocking strategy is to remain the same, the extraction and insertion principals can be used to decompose the modified single block volume grid into the original multiple block format.

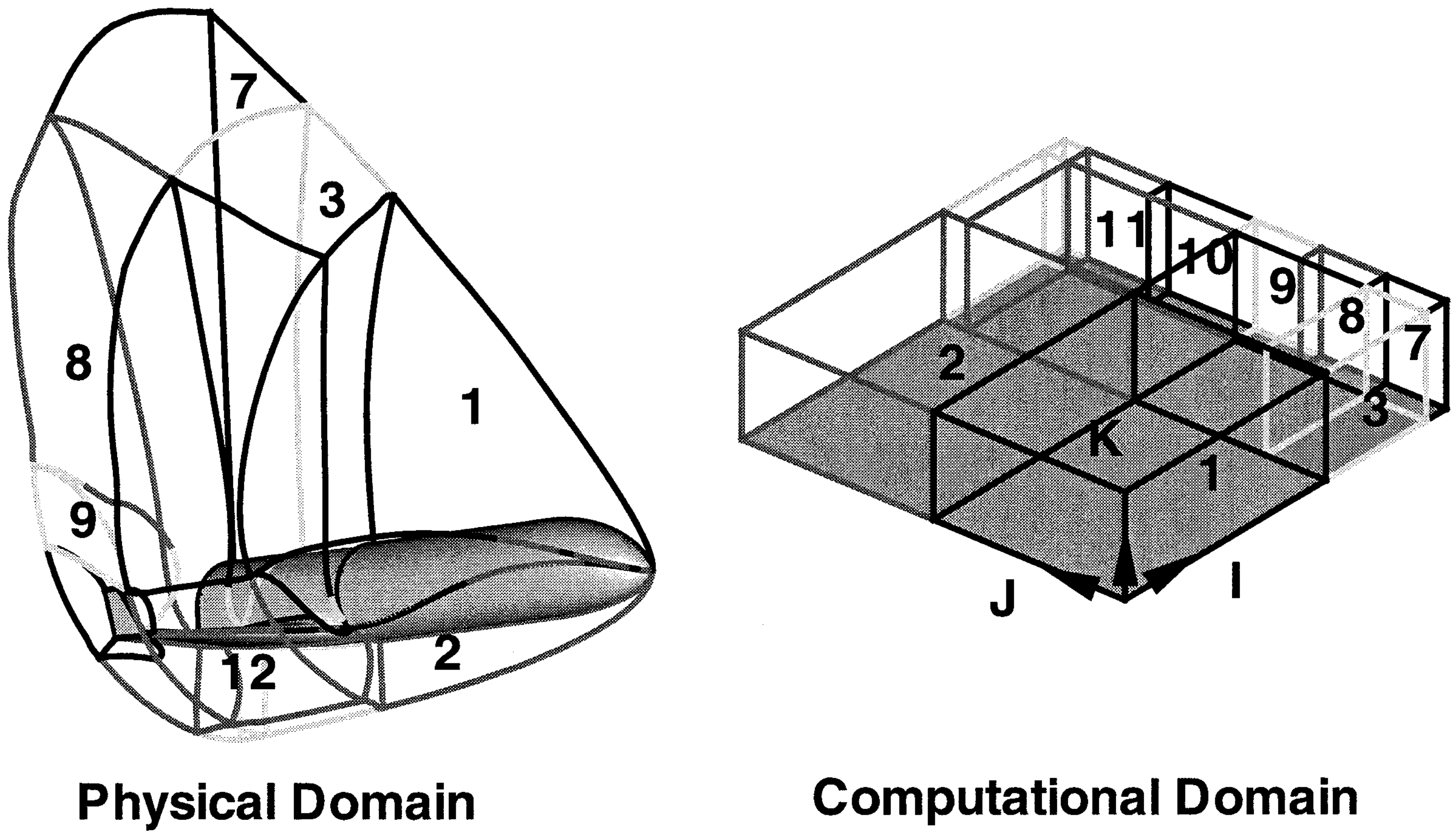

Figure 6.1: Initial multiple block decomposition of NASA proposed SSTO.

6.3 Combining Grid Blocks

If a different blocking strategy is to be used for the above merging example, the grid can be decomposed by using the extraction and insertion methods, then using the **combine** command to generate a new block set. For example, if the New blocking strategy is the following:

Block Number	Computational Limits
1	(161 X 33 X 33)
2	(161 X 33 X 33)
3	(161 X 33 X 33)
4	(161 X 33 X 33)

Table 6.2: New blocking strategy.

The commands to decompose the single block into this multiple block decomposition are:

```
allocate xyz[161,33,33]
allocate xyz[161,33,33]
allocate xyz[161,33,33]
allocate xyz[161,33,33]

#
# Decompose single block:
#

set xyz[3,1] = xyz[2,1,,1-33]
set xyz[4,1] = xyz[2,1,,33-65]
set xyz[5,1] = xyz[2,1,,65-97]
set xyz[6,1] = xyz[2,1,,97-0]

#
# Regroup new blocks into single grid system:
#

combine xyz[3] xyz[4] xyz[5] xyz[6]
```

Script 6.3: Domain decomposition through block splitting.

The results of these commands will generate `xyz[7]` which will contain 4 blocks where each is then referencable as part of the grid system number 7.

Chapter 7

Copying Grids

In the analysis process of evaluating aerodynamic trends in the configuration design process, various changes are made to the configuration to improve vehicle performance. These changes, usually accomplished through parametric design studies, produce volume grids that are very similar both in physical and computational limits. Reuse of a previous computation can reduce the time to compute a new flow field about a modified vehicle definition.

Generating the grid about the new configuration can be difficult. One method is to use a zonal approach[16] which uses various re-distribution techniques to incorporate an elliptically generated parametric design change. Instead of using the various re-distribution techniques explained in ref. 11, copying the original grid spacings into the new elliptically generated grid can accomplish the same task if the parametric design change represents only an incremental modification of the original configuration, as shown in Fig. 7.1.

To copy the grid spacings from one grid into another, use the **copydist** command:

copydist *interpolant basis direction xyz1[...] xyz2[...]*

where,

interpolant is the parameterization to be used for the copy. The possible values for the argument are `arclength` or `normarc`.

basis is the interpolation basis to be used. The possible values can be `linear` or `spline`.

direction is the direction to copy the grid-point distributions, one for each computational index. The possible values are `I-direction`, `J-direction`, or `K-direction`.

xyz1[...] is the source grid block to get the grid-point distributions.

xyz2[...] is the destination grid block containing the grid-lines to be modified.

The **copydist** command does require the number of grid-lines in the first `xyz[...]` to match the grid-lines in the second `xyz[...]` in the two cross-computational directions (i.e. the ones not identified by the *direction* argument). But the number of points in the computational index referenced by the *direction* argument can differ.

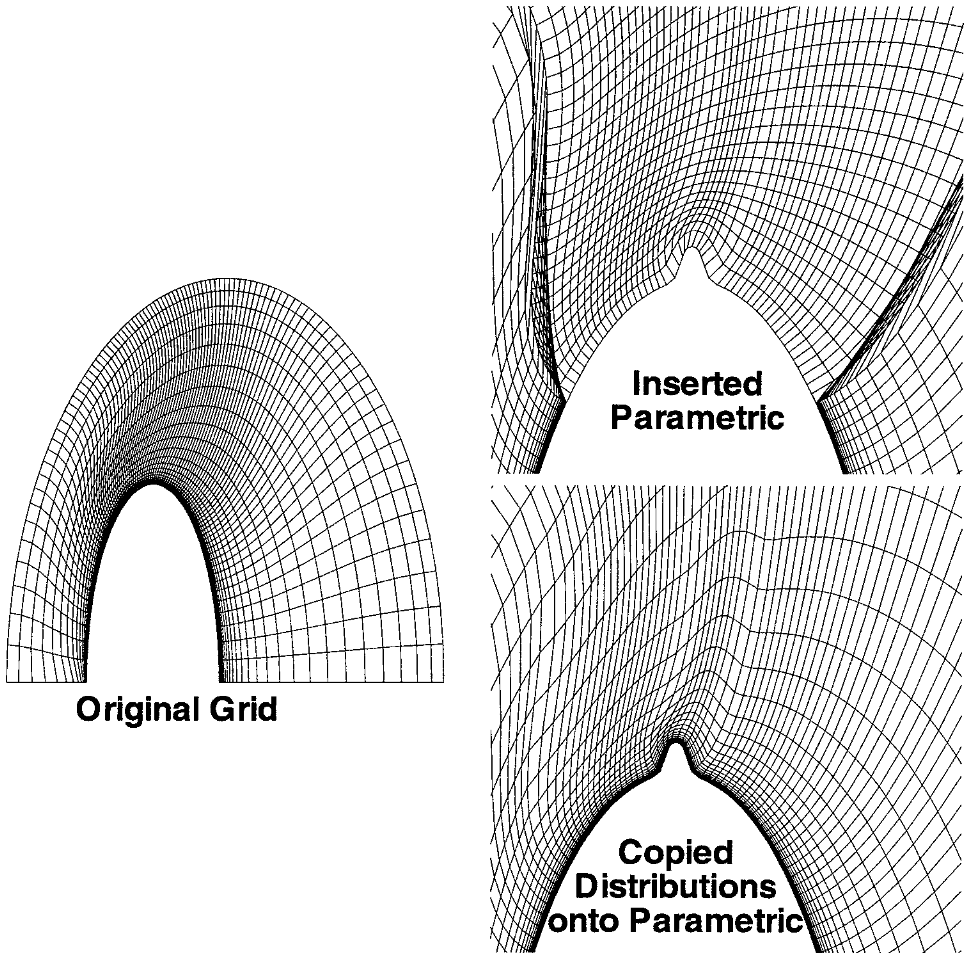

Figure 7.1: Effects of the **copydist** command in normalized arclength domain.

The `spline` basis function is recommended for most of the **copydist** command executions, but the spline function is unclamped which can lead to the generation of negative volumes because of interpolated line over shoot. This occurs because of the way the copying is done. This command copies one grid to another by computing the function of the physical coordinates with respect to the arclength-point distribution on the source and using it as the distribution on the destination grid, as shown in Fig. 7.2. To correct this problem, VGM checks to make sure the arclength function resulting from the `spline` interpolation is never negative. If a region is negative the region is isolated and a linear interpolation is placed between the positive arclengths.

Also in Fig. 7.2, the differences between `linear` and `spline` basis interpolation are also illustrated. In the plot of the physical coordinates, the dashed line is straight between the original grid points while the `spline` is curved. The effect the `spline` has on the grid is very different than the `linear` basis, because the `spline` has a tendency to improve grid line orthogonality at a boundary where that type of PDE solver boundary condition was active. Adapting with `spline` interpolation on a coarse basis grid can improve the orthogonality at a boundary. The `spline` basis is recommended because it usually results in the least amount of change along the grid line from manipulation to manipulation.

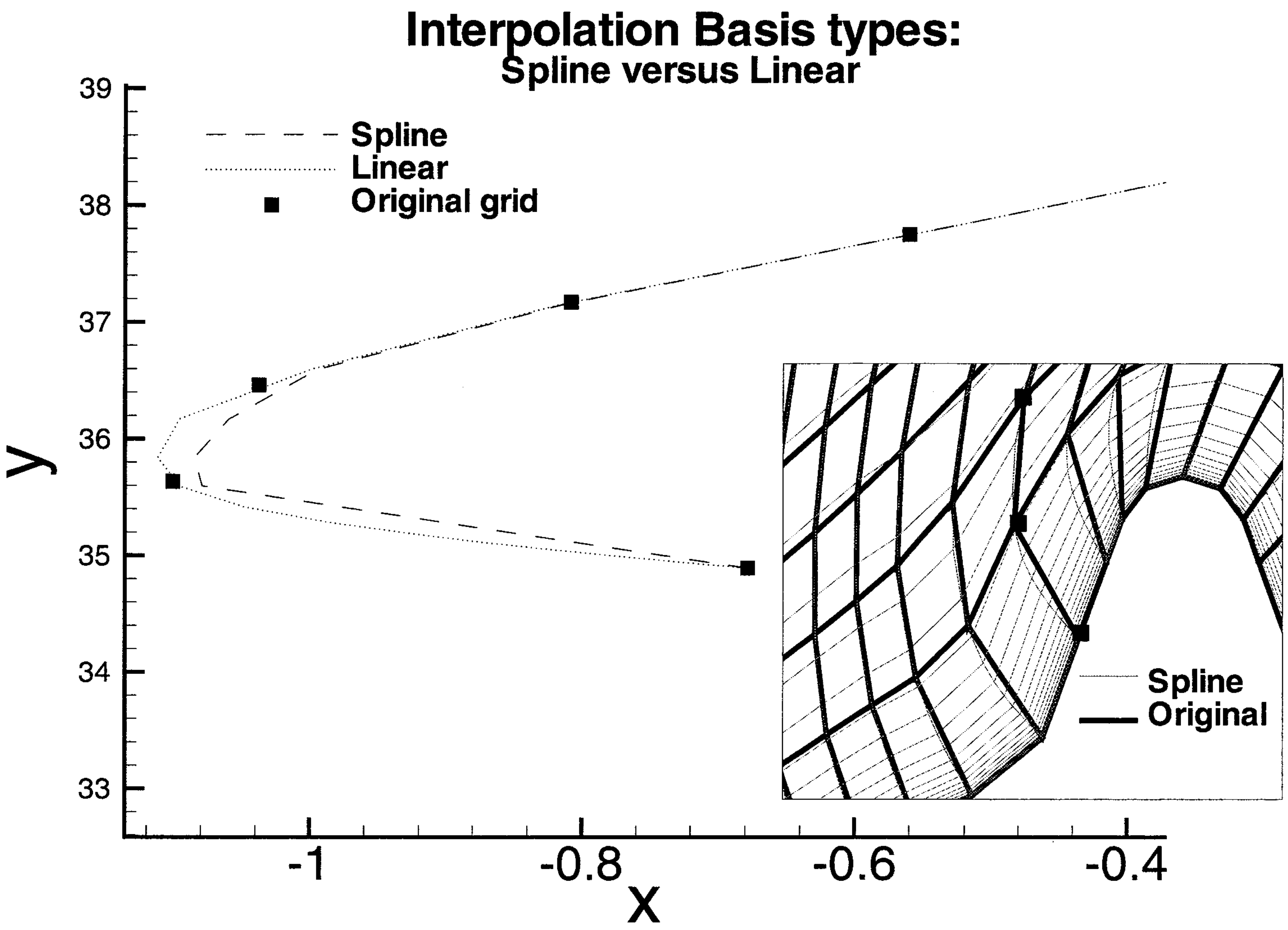

Figure 7.2: Effects of the `linear` and `spline` basis interpolation.

The capabilites of the **copydist** command are explained primarily in this chapter, but the command can use used to:

- Adapt volume grids based on previously grid-adapted, computed flow fields; and
- Inserting one grid into another (see chapter/section 10.1).

7.1 COPYDIST Usage

As stated earlier, one use of the **copydist** command is to insert a vehicle design parametric into an existing volume grid for parametric design studies. To do this, use the following:

```
copydist normarc spline k-direction xyz[1,1,81-0,17-65] xyz[2]
set xyz[1,1,81-0,17-65] = xyz[2]
```

Script 7.1: Insertion of design change for parametric studies.

This command copies the distributions from grid system number 2 in the `k-direction` to the design parametric in the volume grid of grid system number 1. Then the **set** command inserts the adapted vehicle parametric into the original grid to change the vehicle configuration. The results of this copy were shown in Fig. 7.1.

7.2 Manipulative Capabilities of COPYDIST

Besides using **copydist** to augment grid generation processes of parametric design studies, the command finds uses in adapting new grids based on old ones. As stated previously, reusing a previous solution to a similar vehicle or flow conditions as the starting point for another solution can reduce CFD simulation time. The **copydist** command can be used to adapt a new grid based on an old grid. There are two types of adaption that can be done. The first is adaption of a volume grid to change the outer boundary and cluster points based on an old grid. This is accomplished by performing the **copydist** in the `arclength` parameter space:

```
copydist arclength spline k-direction xyz[1] xyz[2]
```

Script 7.2: Utilizing old grid data to adapt a new grid.

The effects of this type of command is illustrated in Fig. 7.3.

The second type of adaption is one based on clustering only. This is done using the normalized arclength (`normarc`) parameter space, and was both explained and illustrated in section 7.1.

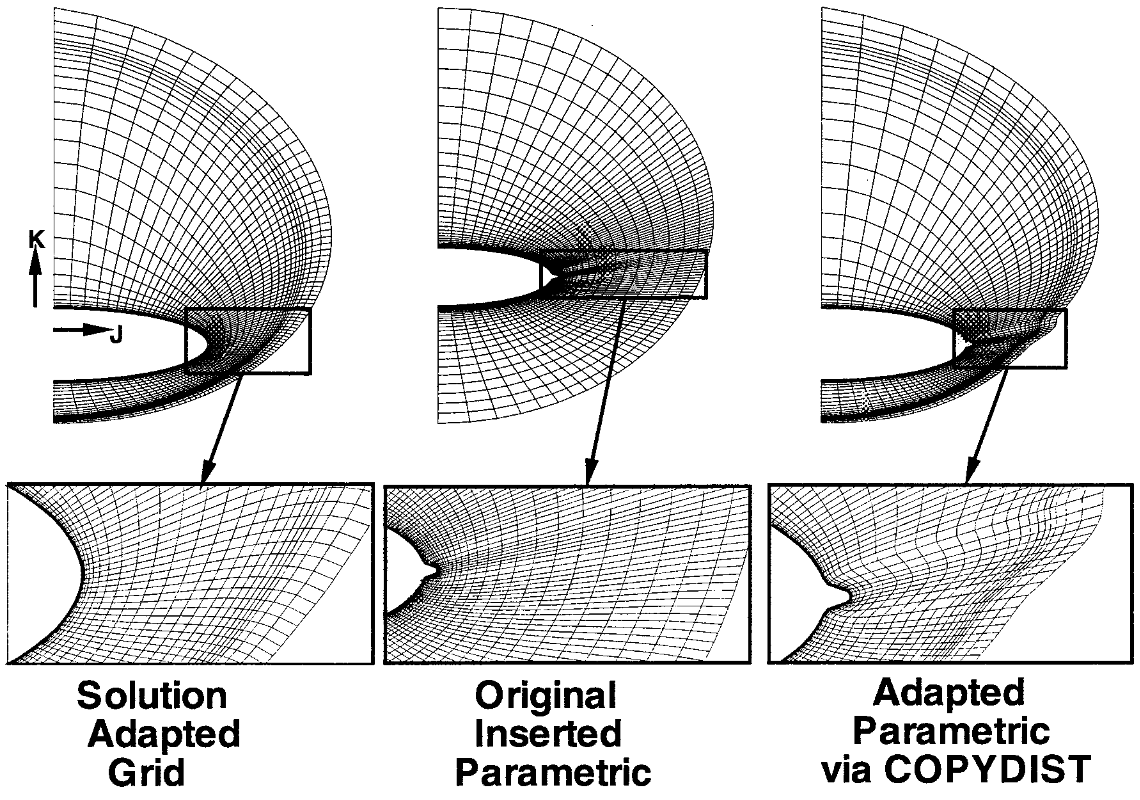

Figure 7.3: Effects of `arclength` parameter used for grid adaption.

Chapter 8

Redistributing Grids

Utilizing the **copydist** command is one method for employing grid reusability techniques. A second technique is to use grid-point redistribution in a computational direction for a group of grid lines. Redistributing a grid line or set of grid lines in a computational direction retains the original grid line along the specified direction, but alters the point distribution. If the grid lines in the redistribution direction have relatively smooth characteristics, the other computational direction grid lines can be modified to improve smoothness or remove poor quality cells and volumes. All redistributing in the VGM code is controlled with the **redist** command. The syntax of this command is:

redist *domain basis direction interpolants points=# distribution_function newblock= xyz[...]*

where,

domain is the physical or computational domain to be used for the redistribution. The possible values for the argument can be `physical` or `parametric`.

basis is the interpolation basis to be used. The possible values can be `linear` or `spline`.

direction is the direction to redistribute the grid-point distributions, one for each computational index. The possible values are `I-direction`, `J-direction`, or `K-direction`.

interpolants is the type of parameterization to be used. The possible values are `arclength` and `normarc` (i.e. normalized arclength).

points=# is the number of points to be generated as a result of the redistribution.

distribution_function this is the function to be used for the redistribution. The functions possible are:

1. `equal`
2. `vinokur`[14](Δs_{begin},Δs_{end})
3. `cubic`(Δs_{begin},Δs_{end})
4. `vin2cub`(Δs_{begin},Δs_{end},ratio)

5. `sin`
6. `-sin`
7. `cos`
8. `laura`(Δs_{begin},S_{max},fstr,ep0,fsh)
9. `func`(*filename*) or `func`(*array_variable*)

newblock= specifies if the results of the redistribution are to be stored in a new grid system and grid block. Possible values are `yes` or `no`.

xyz[...] is the region of a grid block to be redistributed distributions.

The capabilities of the **redist** command are numerous, as this command performs the basis of many types of manipulations, including:

- Copying distributions from one grid to another;
- Adapting grids from a solution adapted grid to a non-adapted grid (see chapter/section 10.1);
- Smoothing grids based on a technique called parametric re-mapping (see chapter/section 9.1);
- Smoothing grids based on retaining cells at limits and placing a smooth distribution function between the fixed cells;
- Converting inviscid grids to viscous grids (see chapter/section 12.4);
- Altering the dimensional limits of a grid while preserving grid quality;
- Improving grid resolution to capture flow field gradients (see chapter/section 10.2); and
- Generating straight line segments (see chapter/section 11.3).

8.1 REDIST Usage

All of the redistributions done with the **redist** command require a distribution function. The VGM code supports eight basic internal functions, and one general function. The internal functions of `equal`, `sin`, `-sin`, and `cos` are based on the grid data at hand. The `equal` function redistributes a grid line to be of equal spacing, while the other three are based on sinusoidal functions. The `sin` function generates a distribution based on the equation:

$$\delta s = S_{\max}\ sin\left[\frac{\pi}{2}\left(\frac{\xi-1}{\xi_{\max}-1}\right)\right]$$

where, ξ is the computational coordinate of a point in the direction of the redistribution. The `-sin` and `cos` functions use the following equations, respectively, for generating a distribution:

$$\delta s = S_{\max}\left\{1 - sin\left[\frac{\pi}{2}\left(\frac{\xi-1}{\xi_{\max}-1}\right)\right]\right\}$$

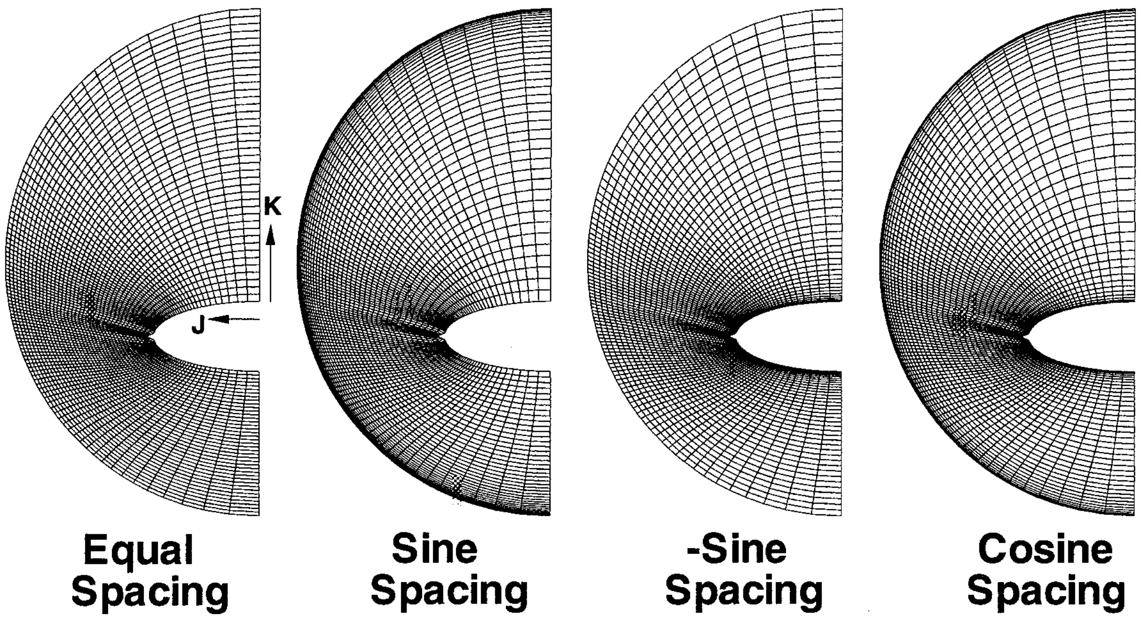

Figure 8.1: Effects of `equal`, `sin`, `-sin`, and `cos` spacing functions.

$$\delta s = \frac{1}{2} S_{\max} \left\{ 1 - cos \left[\pi \left(\frac{\xi - 1}{\xi_{\max} - 1} \right) \right] \right\}$$

The effects of these distributions are shown in Fig. 8.1.

> **NOTE:** With all grid-lines originating from the $K_{\min}$ boundary near the center, the `sin` function clusters points to the $K_{\max}$, the `-sin` function clusters points to the $K_{\min}$, and the `cos` function clusters points at both ends of a grid line.

Illustrated in Fig. 8.2, the cells produced along a computational direction are inversely dependent on the distribution function used (i.e. the `sin` distribution looks like a reversed sine curve from $\pi/2$ to 0). The regions of high curvature on the sine waves produce the smallest change in cell size at the opposite ends to which they are applied. Likewise, the `cos` distribution function produces the smallest cells and largest cell to cell scaling at the end points with near equal spacing on the interior.

Each of these distribution functions find use in computational fluid dynamics by clustering points near a wall for capturing various gradients in a boundary layer with the `-sin` function to improving continuity in grid spacings along a computational direction using the `cos` function. The strength of these distribution functions is that they are not dependent on cell size conditions at the ends of a grid line; they can be used to eliminate such dependency and foster grid quality through such independence.

In some instances, dependency on cell sizes at the ends of grid lines is necessary to promote improved grid quality. The internal functions of `vinokur`, `cubic`, `vin2cub`, and `laura` provide this dependency by requiring input parameters to control the distribution function. The first three require the cell spacings at the beginning and end of a grid line, Δs_{begin} and Δs_{end} respectively, while the `vin2cub` requires a ratio. The ratio is a blending

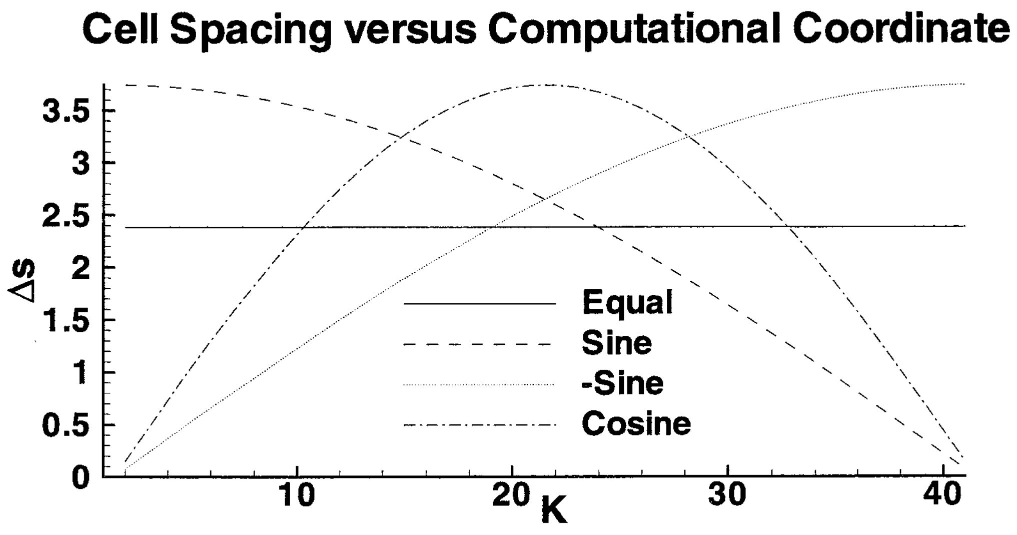

Figure 8.2: Cell to cell scaling effects of `sin`, `-sin`, and `cos` distribution functions.

factor between the `vinokur` and `cubic` functions, using the following equation:

$$\delta s \; = \; (ratio)\delta s_{\text{vinokur}} + (1 - ratio)\delta s_{\text{cubic}}$$

The last internal function generates a distribution based on the ALiGN SHoCK[5] function in the LAURA code, which is used to adapt volume grids based on flow field parameters. The adaption is usually done in the body to shock (outer domain) direction. This function requires the beginning cell height, which is used to obtain a local Reynolds number of approximately 1, an ending total arclength to the bow shock, and three control parameters:

1. *fstr* is the percentage of cells to be placed between the body and the outer bow shock, located at S_{max}.

2. *ep0* controls the amount of clustering at the bow shock (no clustering if this value is zero).

3. *fsh* controls where the bow shock, located at S_{max}, will be with respect to the distance along the grid line from the body to the outer domain.

The effects of these distributions are shown in Fig. 8.3. The input cell sizes were:

$\Delta s_{\text{begin}} = 0.5$

$\Delta s_{\text{end}} = 0.5$

ratio $= 0.5$

$S_{\text{max}} = 47.0$

fstr $= 0.8$

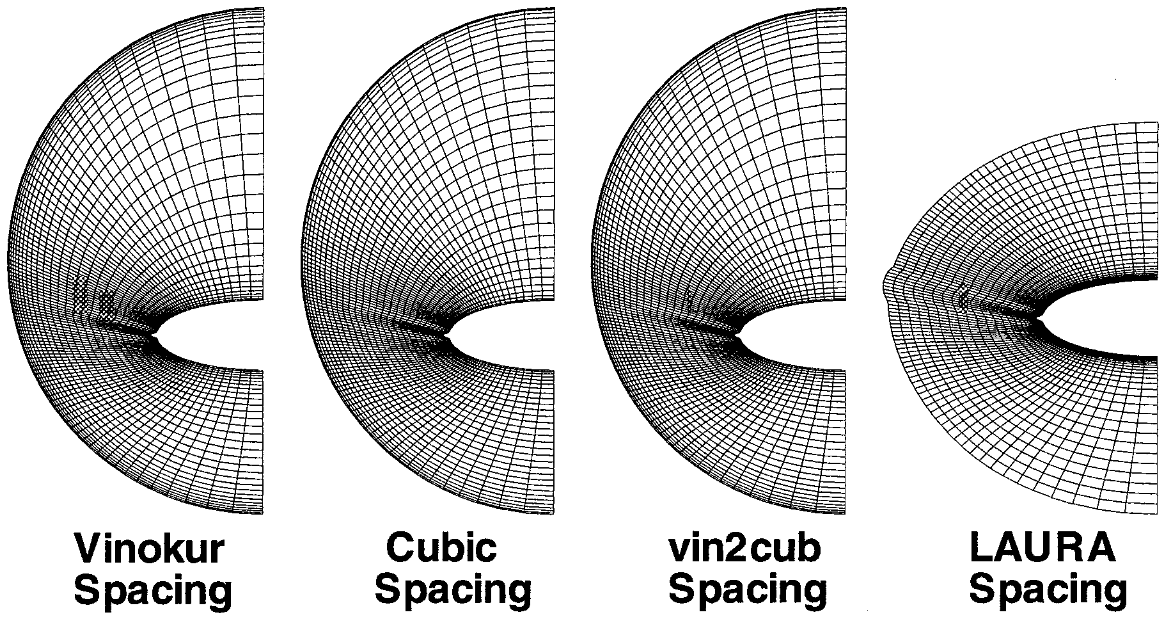

Figure 8.3: Effects of `vinokur`, `cubic`, `vin2cub`, and `laura` spacing functions.

ep0 = 0.0

fsh = 0.0

Each of these spacing dependent functions provide different grid characteristics by the cell to cell scaling of the resulting distibution. Illustrated in Fig. 8.4, the `vinokur` distribution provides nearly equally spaced cell sizes at the end points with considerable stretching on the interior, while the `cubic` distribution exhibits a more subtle stretching on the entire grid line with large changes in cell sizes at the ends. The `vin2cub` distribution blends the characteristics of both the `vinokur` and `cubic` to produce a hybrid of stretching characteristics with reduced cell to cell scaling changes at the ends of the grid line and a moderate stretching on the interior. The `vinokur` distribution is primarily used to provide grid point densities near the wall of a configuration for proper boundary layer modeling and the `vin2cub` function is used to smooth grid lines with poor distributions or large cell to cell scalings that can reduce CFD solver accuracy.[17]

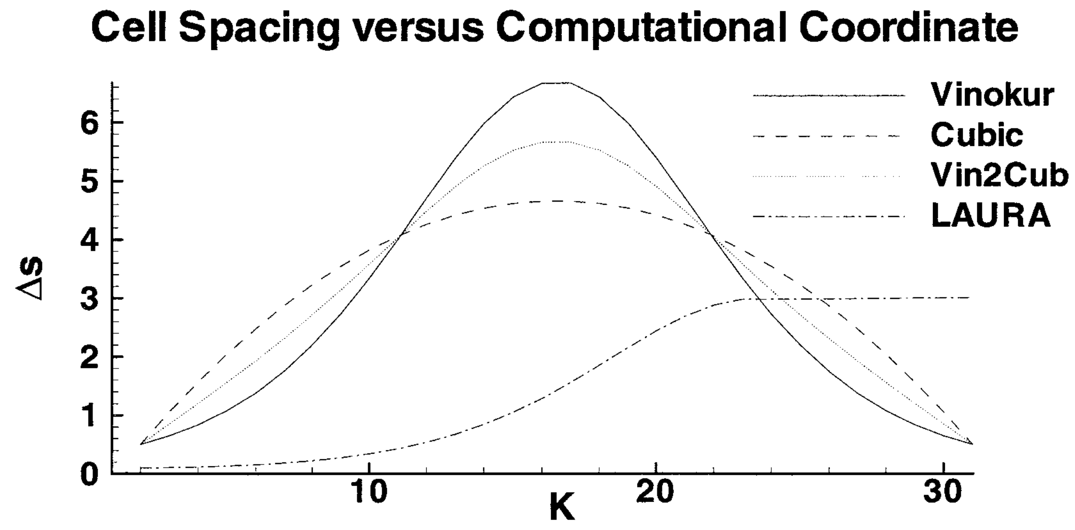

Figure 8.4: Cell to cell scaling effects of `vinokur`, `cubic`, and `vin2cub` distribution functions.

The input values for the internal functions can be constants specified in the distribution function argument, but these values can also be internal constants, array variables or file names. Using the **set** command, constant variables can be extracted from VGM intrinsics to generate the input to the distribution function arguments. For example, to redistribute a grid line using a `vinokur` function but retain the current cell sizes at the end points, the following can be used:

```
allocate dsi1[2,1,1]
set dsi1 = dsia[1,1,45-46,1,0]
set dsi1a = dsi1[2]
set dsi1 = dsia[1,1,60-61,1,0]
set dsi1b = dsi1[2]
redist i-direction spline arclength physical points=17 vinokur(dsi1a,dsi1b) \
        xyz[1,1,45-61,1,0] newblock=no
```

Script 8.1: Grid-line smoothing by fixing endpoints for redistribution.

This set of commands will redistribute a grid line from I=45 to I=61 using the current point spacings at the ends, but place a `vinokur` distribution between the beginning and ending grid points. This is a common manipulation in VGM for smoothing grids, as will be explained in chapter 9. An added feature is the capacity to use array variables to do the same manipulation, but on several cross-direction paired grid lines, (J,K) indexed lines in this case. To do this type, simply replace the `dsi1a` and `dsi1b` with arrays:

```
allocate dsi1[2,31,65]
allocate dsi1a[1,31,65]
allocate dsi1b[1,31,65]
set dsi1 = dsia[1,1,45-46]
set dsi1a = dsi1[2]
set dsi1 = dsia[1,1,60-61]
set dsi1b = dsi1[2]
redist i-direction spline arclength physical points=17 vinokur(dsi1a,dsi1b) \
        xyz[1,1,45-61] newblock=no
```

Script 8.2: Grid-zone smoothing by fixing endpoints for redistribution.

Naturally, these command lines assume the J and K limits are 31 and 65, respectively. These command lines are used to smooth a grid, illustrated in Fig. 8.5, where the thick solid grid lines represent fixed cell size boundaries.

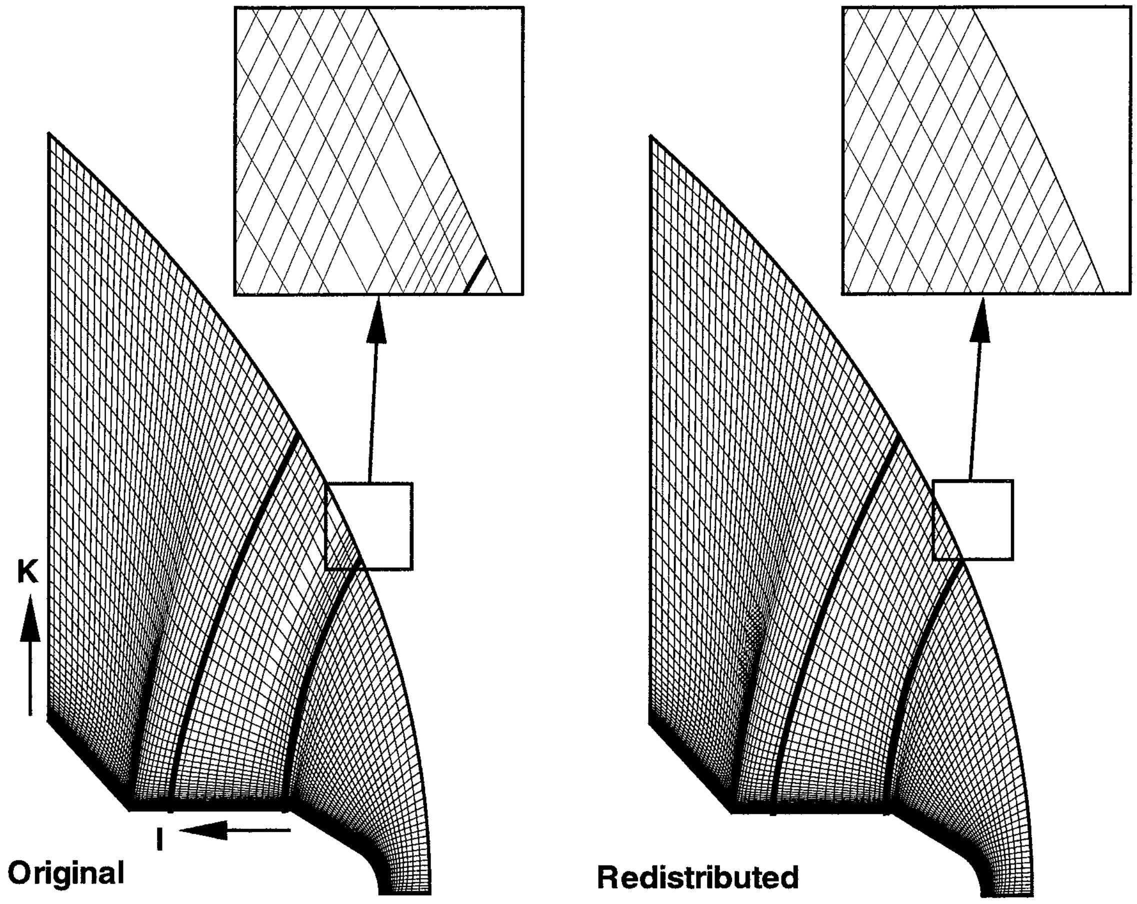

Figure 8.5: Effects of **redist** smoothing of a grid by retaining existing cell sizes.

> **NOTE:** When using array variables as input to the `vinokur`, `cubic`, `vin2cub`, and `laura` distribution functions, the number of cross-direction paired grid lines identified by the `xyz[...]` argument *must* match the limits of each array variable.

One more internal function is available for the **redist** command. This is the `func()` argument. The `func()` argument allows for any distribution function to be used, both internally and externally generated. For an externally generated distribution function, the `func()` requires a file name containing a single function curve in either arclength or normalized arclength, as indicated by the *domain* type in the **redist** command arguments. An example file is shown in table 8.1. The first line indicates the number of points in the function, and the rest of the data is the function in listed form. This format is only used for single line function data. If a surface of data is to be used, it has to be generated internally to VGM.

The `func()` argument can also take an internally generated function. Since the **redist** command redistributed based on a grid line, the function used for the redistribution can be

```
11
0.0000
0.1000
0.2100
0.3200
0.4500
0.6000
0.7300
0.8200
0.9100
0.9500
0.9750
1.0000
```

an arclength parameter. The generation of this type of function is done in chapter 9 because this function is primarily used for smoothing grids.

The arguments in the **redist** command are numerous because this command is the backbone of many possible grid manipulations in VGM. The interpolants, are identical to those explained for the **copydist** command in chapter 7. But an added twist to the **redist** command is the use of different domains to perform the redistribution. The `physical` domain instructs VGM that all parameters used in the distribution function are based on physical grid related values, just as used in the **copydist** command. But the `parametric` domain tells VGM that the distribution function parameters are based on computational domain values, *not* the physical coordinates. For example, in the `physical` domain to keep the current values of cell sizes at the end points of a grid line to be redistributed with a `vinokur()` function, the cell sizes have to be computed using a grid intrinsic. But to specify the same cell size at an end point in the `parametric` domain, the value of 1.0 is used because there is one cell between the end point and the next point onto the interior of the grid line being redistributed. Specifying unity at both end points of the `vinokur()` function in the `parametric` domain will have very little effect because the distribution will be nearly equal in the `parametric` domain.

To harness the power of the `parametric` domain redistribution, a non-unity value at an end point will change the parametricity of the domain being used for redistribution, but will not significantly change the overall distribution. Only the density of the grid points at an end will change. For example, a grid can be adapted so that the overall grid line character remains the same but the clustering or grid point density is increased at one end in a flow field. For instance, at the wall of a configuration. The effects of specifying `physical` and `parametric` domains is illustrated in Fig. 8.6.

Notice that the parametric grid retains the same grid line character in the cross-sectional direction, but the points are more clustered at the wall of the configuration. The `physical` domain redistribution changed the distribution function and the point densities to be equal at all beginning and ending points. Also notice that the grid line curvature in the J-direction in the `physical` domain redistribution is more discontinuous than the `parametric`. Utilizing `parametric` domain redistribution in the K-direction retains the grid-line curvature in the cross-directions thereby only modifying grid-point densities in the redistribution direction.

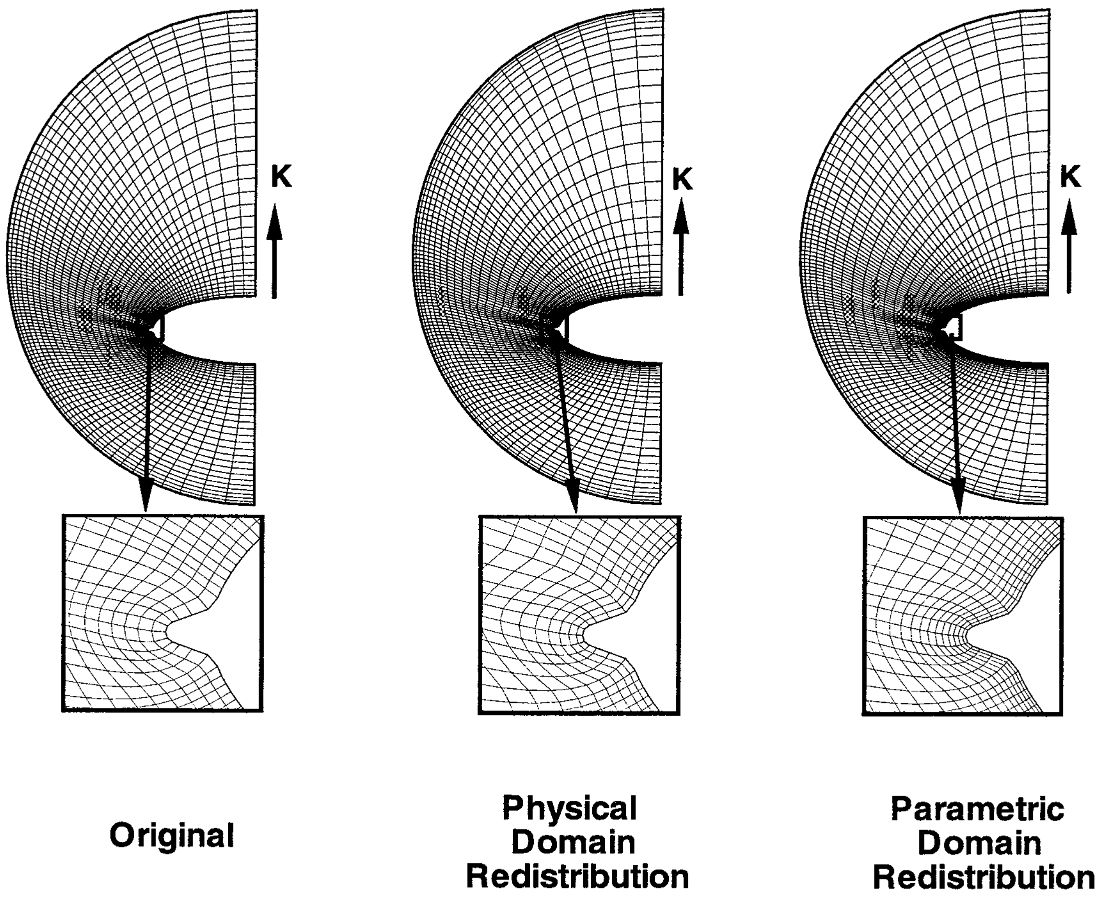

Figure 8.6: Effects of `parametric` and `physical` domain redistributions for grid adaption.

This example illustrates the strength of the `parametric` redistribution. These redistributions were produced by using the following VGM commands:

```
redist j-direction spline arclength physical points=31 vinokur(.25,1.)  \
       xyz[1] newblock=yes

redist j-direction spline arclength parametric points=31 vinokur(.25,1.)  \
       xyz[1] newblock=yes
```

This discussion describes the basic manipulative capabilities of the **redist** command. More powerful capabilities are incorporated into separate sections in the chapters on smoothing (chapter 9), and grid adaption (chapter 10).

Chapter 9

Smoothing Grids

The VGM code was initially designed to do the insertion of design parametrics into existing volume grids for parametric design studies. During this process, a zone is inserted and blended into the original volume grid that was used to define the region to be modified. In so doing, there are many instances in which grid smoothing is required, especially to resolve grid point spacing mismatches that may have been generated in the development of a design parametric volume grid.

There are other instances where grid smoothing may be necessary, including:

- Highly oscillatory grid lines in one direction resulting from poor elliptic solver boundary conditions;
- Open volumetric spaces due to inadequately defined definition boundaries;
- Previous manipulations correct one problem but create another;
- Kinked grids resulting from poor adaptions to flow gradients; and
- General improvement of existing grids.

Since VGM is a manipulation language that contains numerous commands, there are many different methods available for smoothing volume grids. The types of grid smoothing that will be explained in this chapter include:

- Parametric Re-mapping - changing of the arclength parameter space with dependent or directionally dependent functions;
- Trans-Finite Interpolation
- Vector Interpolation - interpolates vectors computed from derivatives derived from existing grid lines.

9.1 Parametric Re-mapping

The underlying technique that will be exploited for Parametric Re-mapping, is the notion of using the arclength parameter as a distribution function for manipulating the location of grid points. This is done by computing the arclength function in a computational direction using equation 9.1:

$$\Delta S_{i,j} = \sqrt{\Delta x_{i,j}^2 + \Delta y_{i,j}^2 + \Delta z_{i,j}^2} \tag{9.1}$$

where,

$$\Delta x_{i,j} = x_{i,j} - x_{i,j-1}$$

$$\Delta y_{i,j} = y_{i,j} - y_{i,j-1}$$

$$\Delta z_{i,j} = z_{i,j} - z_{i,j-1}$$

if the direction of the arclength is J. The initial function, when mapped to the physical domain, locates the grid points in their original positions but creates a bridge between the coordinates. This will be referred to as the basis function. By changing the distribution function for the length of the curve that passes through the grid points in the said computational direction, the grid point locations along the basis function (i.e. curve) will be changed. The arclength parameter is one dimensional, which offers a simple link to the three dimensional physical domain. By grouping a series of arclengths to form a region (i.e. zone) the new distributions used for the arclengths in a computational direction can be created with a single function or multiple dependent functions. The modification of the arclength parametric domain through the use of single or multiply dependent functions to create new distribution functions, is termed parametric re-mapping.

The **copydist** and **redist** commands, by virtue of their arguments and implementation, change the distribution of grid points with single functions. But the **redist** command has an added feature, the `func()` argument. This argument can take a computational zone of arclength parameters in a single direction to define the new distribution functions to be applied to a surface or volume grid. To generate the computational zone of arclength parameters, an initial domain needs to be defined using the **allocate** command. Then, using one of the intrinsics in table 5.1, an arclength parameter is computed in a computational direction for the **allocate**d region. Now a new command is used to develop a dependent function across the region in one or both directions computationally orthogonal to the direction used to compute the arclength parameter. This command is called **blend** and the syntax is as follows:

blend *varname[I-limit,J-limit,K-limit] direction dimension domain interpolation=* {*xyz[...]*}

where,

varname is the core variable containing an arclength parameter to be blended for smoothing a grid.

direction is the direction to blend the arclength parameters. The possible values are `I-direction`, `J-direction`, or `K-direction`.

dimension is the dimension of the blend. This can be either:

1. `1d` - single dimension
2. `2d` - two dimensions
3. `3dp` - two dimensions but planar by stepping through the third dimension

4. `3dw` - three dimensions

domain is the physical or computational domain to be used for the blending. The possible values for the argument can be `physical` or `parametric`.

> **NOTE:** If the `parametric` domain is used, the parameterization of the domain being **blend**ed is based on the computational coordinates, and the `xyz[...]` need not be specified. Conversely, if the `physical` domain is to be used, the `xyz[...]` argument *must* be present and *must* have the same dimensions of the variable being **blend**ed.

interpolation= this is the interpolation scheme to be used to blend from one know index to another. The schemes possible are:

1. `linear`
2. `elliptic`[18]
3. `spline`
4. `tfi`
5. `larcs(#,#,#)`[19]

> **NOTE:** The `linear`, `elliptic`, and `spline` interpolation schemes are only available in one dimensional interpolation; the last two are for `2d`, `3dp` and `3dw` interpolation.

xyz[...] is the region of a grid block to be used for computing arclength blending functions if the *domain* is `physical`.

The capabilities of the **blend** command include:

- Iterative grid smoothing through the altering of basis distribution functions;
- Grid smoothing through the use of parametric re-mapping;
- Blending from a solution adapted grid to a non-adapted grid; and
- Adapting grids based on a solution adapted grid (see chapter/section 10.1).

9.1.1 One-Dimensional Parametric Re-mapping

This command operates on a single grid based quantity, such as the internal intrinsics. The command performs the blending by using the *interpolation* scheme to blend between boundaries of known data. For example, to linearly interpolate an outer boundary of a volume grid, the arclength function on a specified region is computed using the `dska` intrinsic, in the wall to outer domain direction in Fig. 9.1.

As shown in Fig. 9.1, the forebody grid has been adapted using some unknown scheme. The aftbody is not adapted because it has not been computed, but the interface between the fore- and aftbodies is highly discontinuous. To correct this problem and **blend** between the adapted and non-adapted grids, a blending region is identified, and the arclength parameter in the wall to outer domain direction is computed. To blend between the adapted

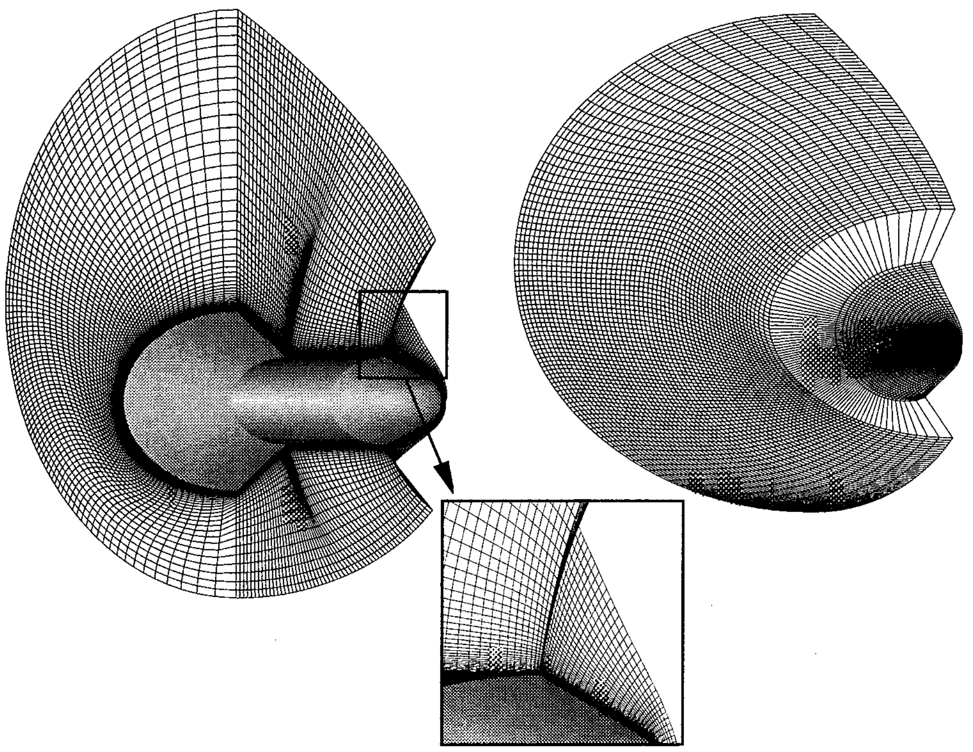

Figure 9.1: Initial outer domain of an adapted and non-adapted grid.

and non-adapted grids, the endpoints of the region are fixed in the streamwise direction (`i-direction`) and blending occurs on the K-direction based arclengths in the `i-direction`. Two types of domain dependent blendings are available, computational and physical. The blending is done by fixing the endpoints at the I-limits, and linearly blending between the endpoints to determine (i.e. compute) the interior K-direction based arclengths. This is done by specifying the limits in the *varname[...]* as 1-0:0 for the I-direction. The new K-direction based arclengths are used to alter the existing volume grid using the `func()` argument of the **redist** command because the new arclengths are the new dependent distribution functions for the identified region in the K-direction. To apply these new arclengths, the following VGM commands were used for the `physical` domain blending:

```
allocate dsk1[33,91,33]
set dsk1 = dska(xyz[1,1,41-71])
blend dsk1[1-0:0] 1d i-direction physical interpolation=linear \
        xyz[1,1,41-71:0]
redist k-direction spline arclength physical points=33 func(dsk1) \
        xyz[1,1,41-71:0] newblock=no
```

Script 9.1: Physical domain based grid smoothing by parametric re-mapping in 1D.

and for the computational domain based blending:

```
allocate dsk1[33,91,33]
set dsk1 = dska(xyz[1,1,41-71])
blend dsk1[1-0:0] 1d i-direction parametric interpolation=linear \
        xyz[1,1,41-71:0]
redist k-direction spline arclength physical points=33 func(dsk1) \
        xyz[1,1,41-71:0] newblock=no
```

Script 9.2: Computational domain based grid smoothing by parametric re-mapping in 1D.

Illustrated in Fig. 9.2 are the results of both domain interpolant types. Though not clear in this example, there is a difference between computational and physical domain based interpolants. The computational domain interpolants are best when the grid is uniformly spaced, as in Fig. 9.2. If the grid is non-uniform in the direction of the **blend** the computational based interpolants will cause dramatic changes in the blending because of the missing dependency on the grid. On the other hand, the physically based interpolants are well suited for both non-uniform and uniform spacing because of the added dependency on the grid physical coordinates. For example, if the blending region of a grid is that of Fig. 8.5, and the region is blended from the wall to the outer boundary (i.e. `k-direction`), the results are shown in Fig. 9.3.

Clearly, the physically based arclengths are best suited to this problem, but only because the cell spacings in the K-direction are non-uniform. Uniform spacing in the blending direction should result in nearly identical results to the computational domain based interpolants, as shown earlier. This described process is known as parametric re-mapping because the K-direction based arclengths were re-computed using a blending function that inherently makes

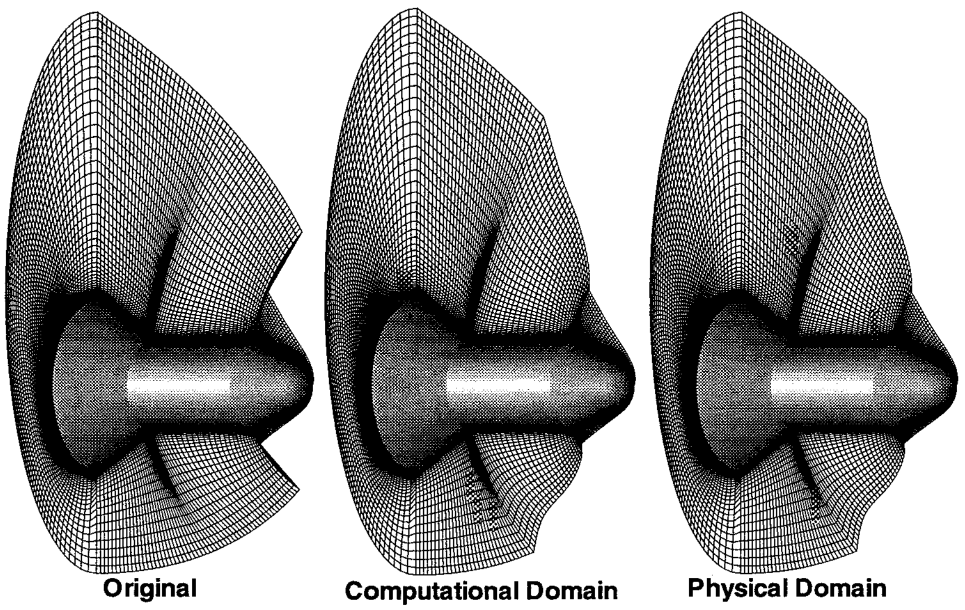

Figure 9.2: Domain based blending for smoothing/adapting a grid.

all the grid lines dependent upon one another; thereby re-mapping the arclength parameter.

The parametric re-mapping process is not limited to linearly based interpolation. For the above example, an elliptic blending function could also be used. Illustrated in Fig. 9.4, are the differences between `linear` and `elliptic` interpolation schemes. Notice that the `elliptic` scheme has more continuous grid lines in the blending direction than the `linear`. This is primarily due to the elliptic function having C-I and C-II continuous derivatives at the endpoints.

This effect of continuous derivatives at the endpoints can be further exploited for smoothing grids. For example, in Fig. 9.5, the grid lines in the surface grid are highly oscillatory (i.e. kinked).

Smoothing this grid by simply applying the elliptic scheme of the **blend** command on a region encompassing the poor grid line characteristics should be sufficient. Note that K-direction grid lines are fairly straight with respect to the I-direction. Due to the straightness of the K-direction lines, these lines will be used to compute the arclength parameters for smoothing, and the blending will be done in the I-direction. To do this, the following VGM commands are used:

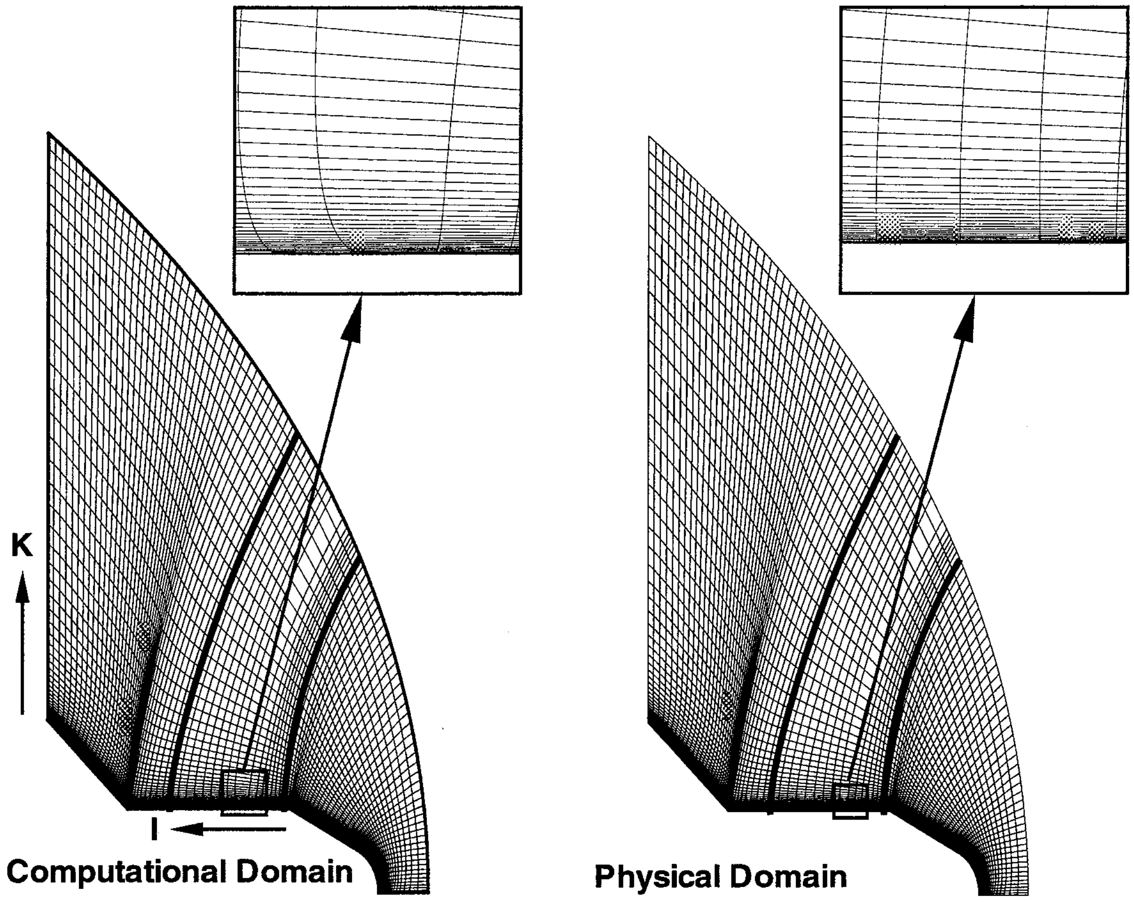

Figure 9.3: Domain based blending for blending a grid.

```
allocate dsk1[7,1,61]
set dsk1 = dskn(xyz[1,1,28-34,1])
blend dsk1[:0] 1d j-direction parametric interpolation=elliptic
redist k-direction spline normarc physical points=61 func(dsk1) \
       xyz[1,1,28-34,1] newblock=no
```

Script 9.3: Algebraic grid smoothing with elliptic coefficients in 1D.

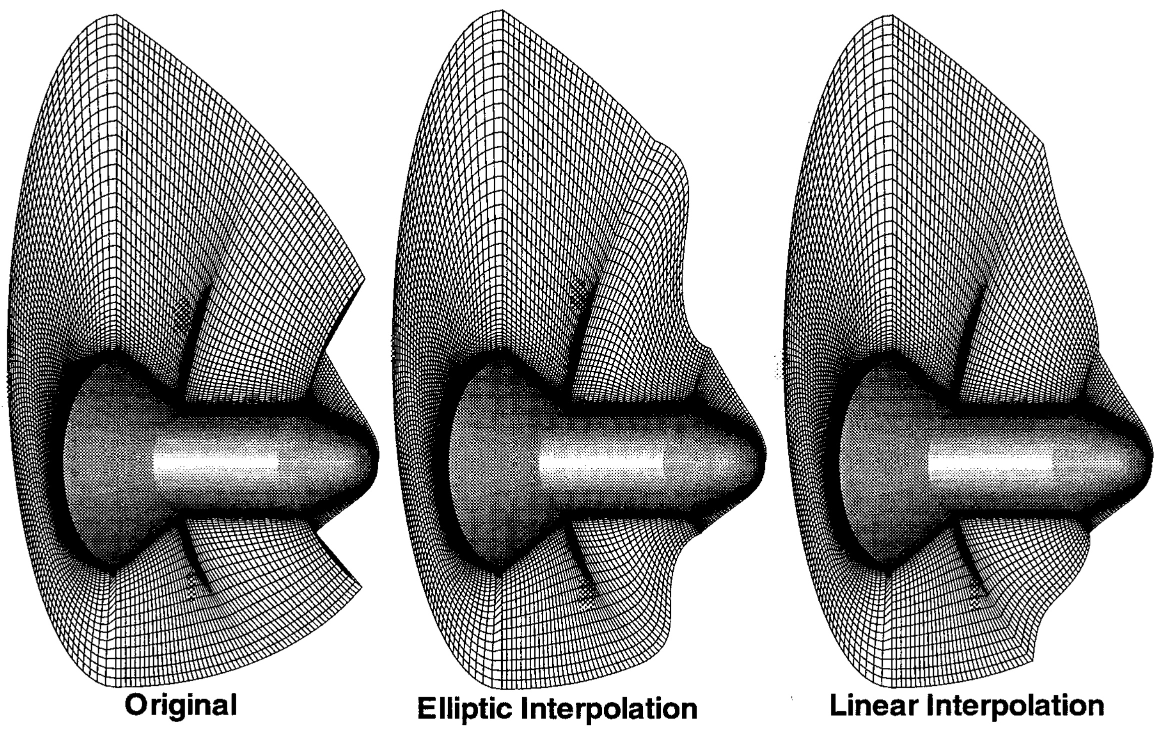

Figure 9.4: Linear and elliptic interpolation for smoothing/adapting a grid.

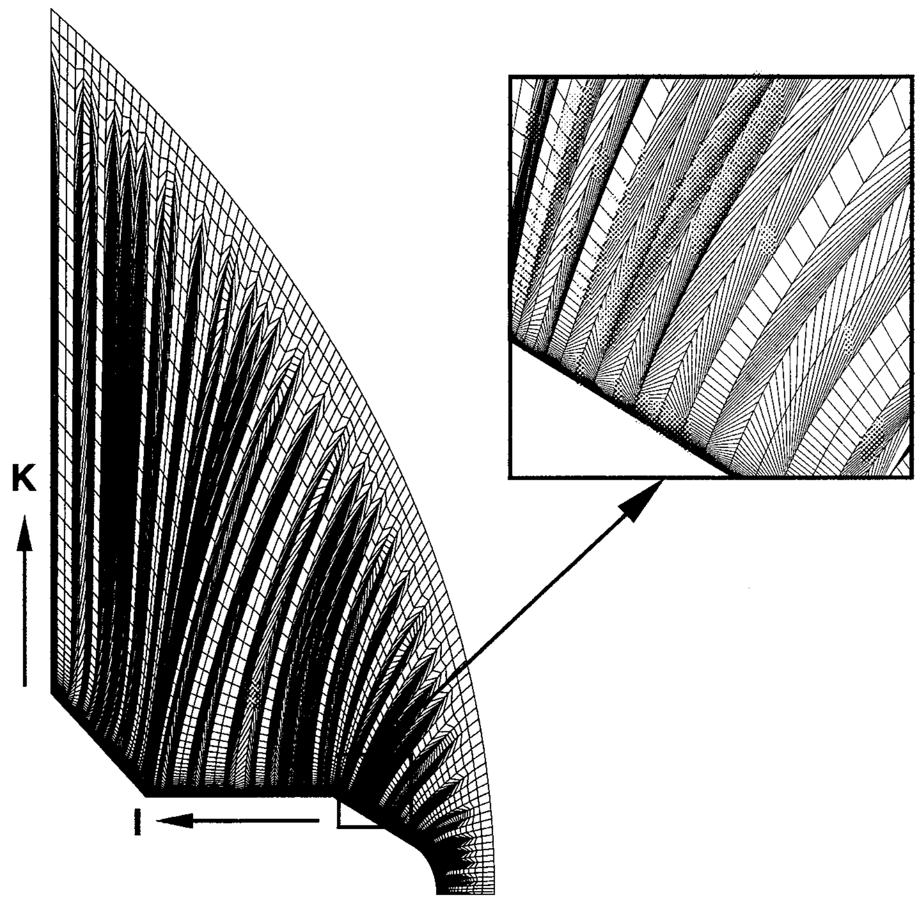

Figure 9.5: Kinked grid lines from poor adaption parameters.

The resulting smoothed grid is shown in Fig. 9.6.

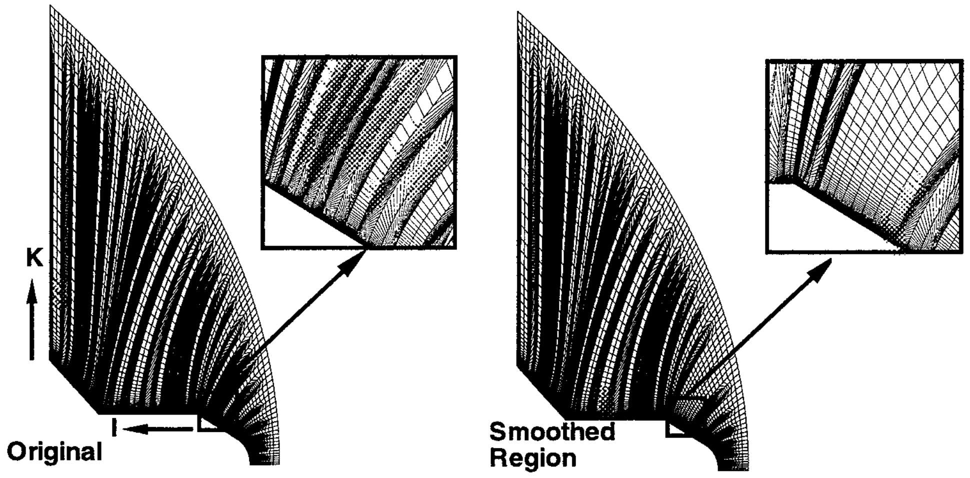

Figure 9.6: Elliptic blending function used to smooth kinked grid lines in a specific region.

The increment on the limits do not always have to be the maximum. If a surface is to be broken into multiple regions, the increment can be less than the maximum. If multiple regions are used, the `spline` scheme can be used to smooth surface and volume grids, by simply fixing intermediate curves or surfaces to be used in the control of the blending. For the above grid, if the increment is set to 5, the smoothed grid appears as shown in Fig. 9.7.

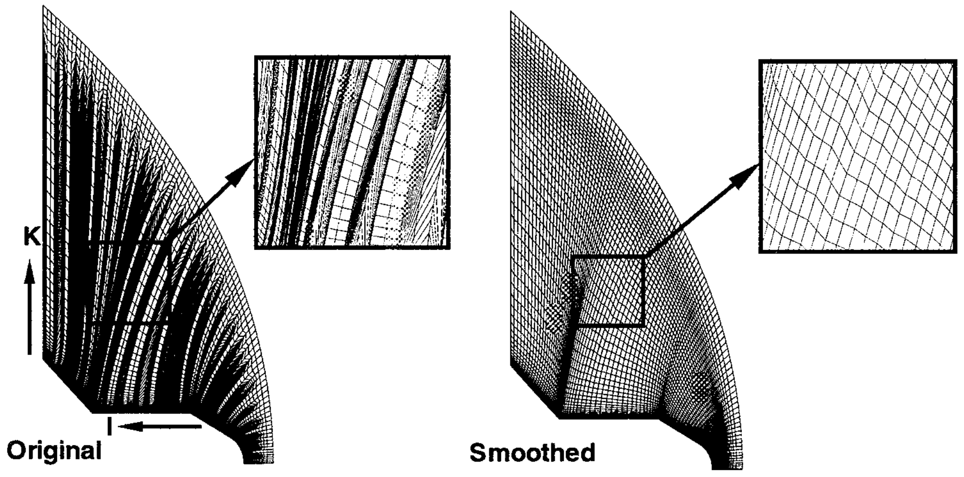

Figure 9.7: Spline blending function used to smooth kinked grid lines in the entire grid.

Although this grid is smoother than the original grid, it still exhibits some grid discontinuities on the interior. These can be removed by iteratively applying the parametric re-mapping, because each time the grid is redistributed the basis functions will be different. Usually 2 or 3 passes is adequate. For the above example, two passes of the `spline` interpolation with multiple regions were done, resulting in the grid shown in Fig. 9.8.

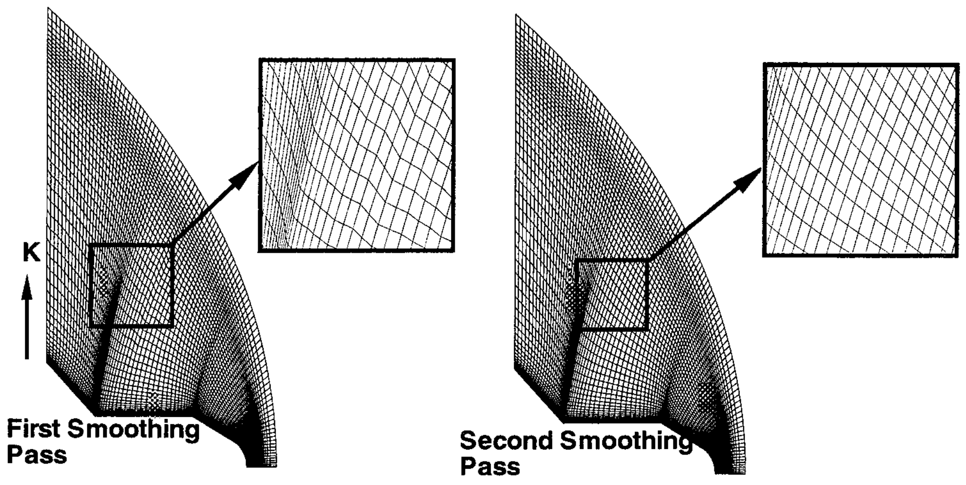

Figure 9.8: Iterative smoothing of kinked grid lines with the `spline` blending function.

The VGM commands used to iteratively smooth this grid were:

```
allocate dsk1[101,1,61]

# Iteration 1:

set dsk1 = dskn(xyz[1])
blend dsk1[:5] 1d i-direction physical interpolation=spline \
        xyz[1,1,:5,1]
redist k-direction spline normarc physical points=61 func(dsk1) \
        xyz[1] newblock=no

# Iteration 2:

set dsk1 = dskn(xyz[1])
blend dsk1[:5] 1d i-direction physical interpolation=spline \
        xyz[1,1,:5,1]
redist k-direction spline normarc physical points=61 func(dsk1) \
        xyz[1] newblock=no
```

Script 9.4: Iterative algebraic smoothing via basis function manipulation in 1D.

A secondary reason this iterative method works is that the `spline` function creates a dependency on each zone, as identified by the non-maximum increment in the I-direction. Other multiple zonal approaches will be discussed, but they do create a dependency from zone to zone, and are therefore not worth applying to iteratively smooth a grid.

9.1.2 Two-Dimensional Parametric Re-mapping

Another scheme could have been used to smooth these kinked grid lines, the two dimension trans-finite interpolation, as detailed in the following VGM commands:

```
allocate dsk1[101,1,61]
set dsk1 = dskn(xyz[1,1,,1])
blend dsk1[:5,,:0] 2d j-direction physical interpolation=tfi \
        xyz[1,1,:5,1,:0]
redist k-direction spline normarc physical points=61 func(dsk1) \
        xyz[1,1,,1] newblock=no
```

Script 9.5: TFI smoothing of distribution functions in 2D.

or through the use of LARCS interpolation:

```
allocate dsk1[101,1,61]
set dsk1 = dskn(xyz[1,1,,1])
blend dsk1[:5,,:0] 2d j-direction physical interpolation=larcs(2,2,2) \
        xyz[1,1,:5,1,:0]
redist k-direction spline normarc physical points=61 func(dsk1) \
        xyz[1,1,,1] newblock=no
```

Script 9.6: LARCS smoothing of distribution functions in 2D.

> **NOTE:** The direction of interpolation is (J). This is due to the plane being a constant J-surface in an existing volume grid. The limits have also changed; the I- and K-limits are the maximum. These limits tell VGM which *varname* function to compute, and which boundaries are to be held fixed. For the two dimensional interpolation, the specified limits must have two computational directions to be determined.

The trans-finite scheme is well documented in other literature, but the LARCS usage can be found in the 3DMAGGS manual. Basically the LARCS method uses `linear` or `elliptic` blends between opposing edges and then uses either of the blends between the edges as the new computed arclengths, or a hyperbolic combination of both opposing edge paired blends. The LARCS scheme is controlled through the 3 numbers in the parentheses:

- Number 1: Linear (1) or Elliptic (2) blend between opposing edges of the first computational index in the right handed coordinate system of the surface to be blended;
- Number 2: Linear (1) or Elliptic (2) blend between opposing edges of the second computational index in the right handed coordinate system of the surface to be blended;
- Number 3: First opposing pair (1), second opposing pair (2) or a hyperbolic combination of both blends (3).

Illustrated in Fig. 9.9 and Fig. 9.10 are the results of trans-finite and LARCS interpolation, respectively. The LARCS method used was the blending of arclength parameters between

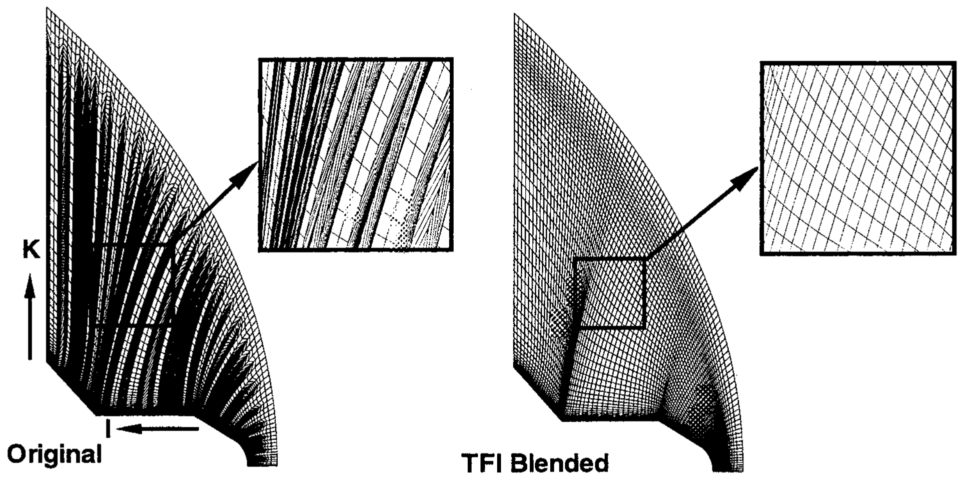

Figure 9.9: TFI blending function used to smooth kinked grid lines in two dimensions.

I-bounded zone edges because that direction was more indicative of they type of smoothing needed. To use the hyperbolic blend of both would produce unfavorable results as the only values being blended are 0.0 to 1.0, (i.e. the normalized arclengths). The hyperbolic blending would take 50% of both blended opposing edge functions at the middle of each boundary edge, thereby increasing the value of the normalized arclengths in the K-direction and cause the grid to oscillate more regularly. The blended edges of the first opposing pair, in the K-direction, would exaggerate the oscillatory grid lines in the I-direction; eliminating its usage.

Notice the slope continuity in the TFI blended grid versus the LARCS blending. The TFI algorithm in this case produced an acceptable grid, while the LARCS did not, because it still has some kinks. This may be due to the fact that the LARCS methods do not have the correcting terms as the TFI methods do. Irregardless, the LARCS methods are available as an alternative to TFI. Consecutive iterative manipulations with the TFI or LARCS methods will not improve the grid because there is no dependency from zone to zone, as identified by the non-maximum increments in the blending direction.

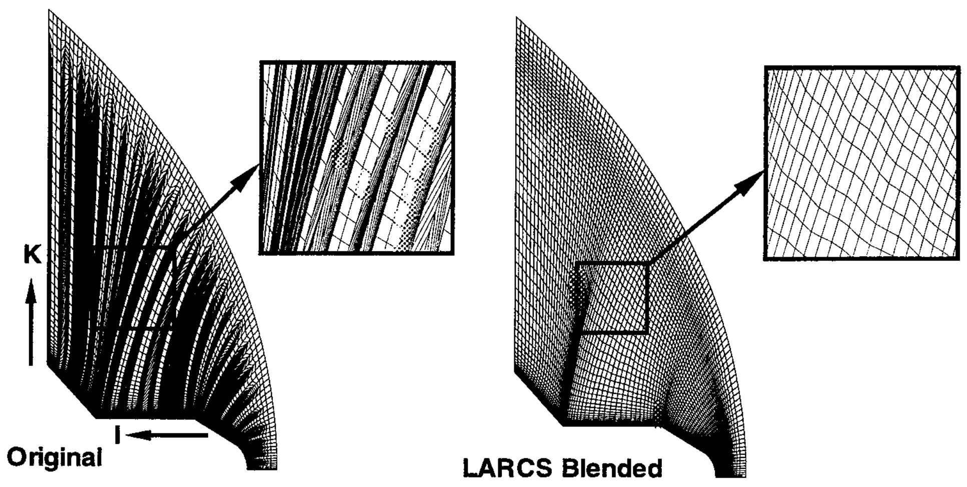

Figure 9.10: LARCS blending function used to smooth kinked grid lines in two dimensions.

9.1.3 Three-Dimensional Parametric Re-mapping

In most cases, a one-dimensional or two-dimensional blend is adequate for smoothing surface and volume grids. There are some instances where three-dimensional blends can produce the most favorable results. Such instances usually occur where slope continuity is a must at multiply bounded regions, as opposed to two or four bounds for the one-dimensional and two-dimensional blends, respectively. In these rare instances, the `3dp` and `3dw` interpolation schemes can be used. For example, the results of running the 3DMAGGS code on a sphere-cone are shown in Fig. 9.11. Notice that the wall to outer domain grid lines are relatively straight, but the J-directional lines are highly curved, just before the frustum. Here, a one-dimensional blend in the J-direction may not produce the most favorable results because the grid will no longer be dependent in the I-direction. To ensure a continuous dependency on the grid, applying a trans-finite interpolation on the K-direction arclengths in the I- and J-directions for each successive plane should be best. To do this type of plane-by-plane interpolation the `3dp` interpolation scheme is used:

```
allocate dsk1[56,363,33]
set dsk1 = dska(xyz[1,1,1-56])
blend dsk1[1-0:0,1-0:0] 3dp k-direction physical interpolation=tfi \
        xyz[1,1,1-56,:0,:0]
redist k-direction spline arclength physical points=33 func(dsk1) \
        xyz[1,1,1-56] newblock=no
```

Script 9.7: TFI smoothing of distribution functions in computationally 3D planes.

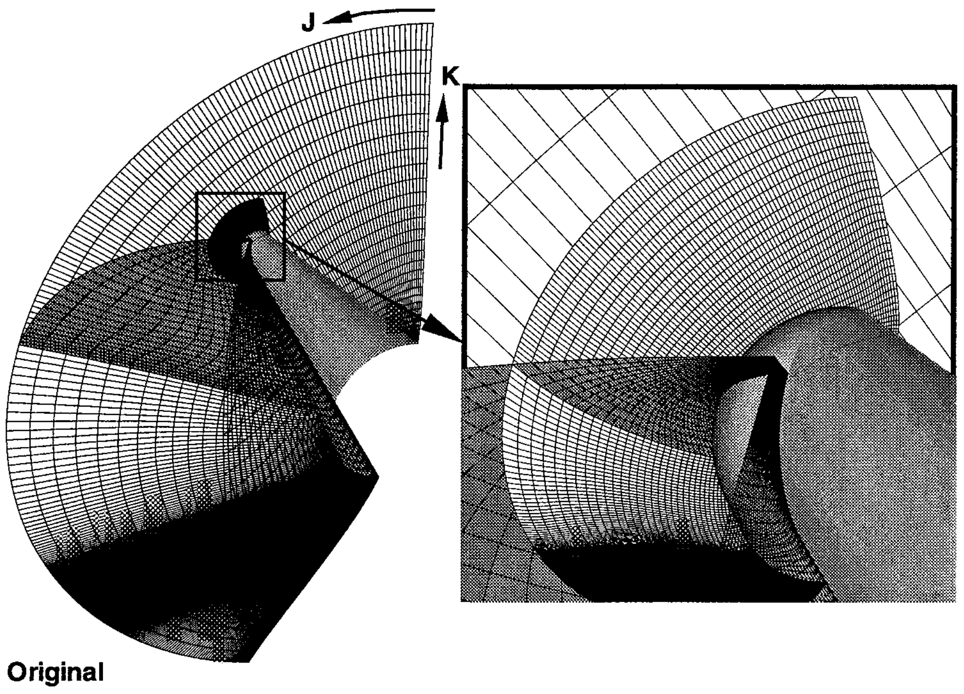

Figure 9.11: Poor elliptic PDE orthogonal boundary condition at a symmetry plane.

The effects of this interpolation scheme are shown in Fig. 9.12. The resulting volume

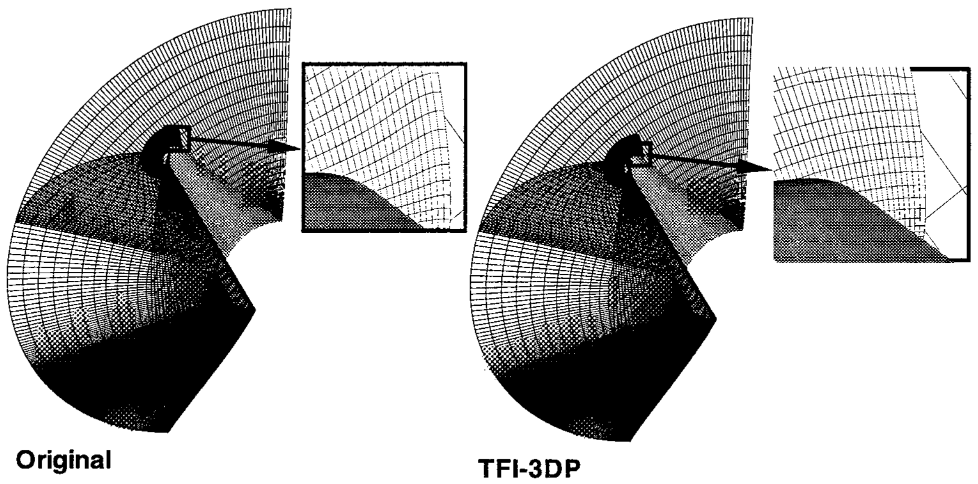

Figure 9.12: Improved boundary condition at a symmetry plane through the `3dp` blending.

grid maintains some consistency between the modified grid and the unmodified grid, as well as improving the orthogonality at the symmetry planes. Use of this type of manipulation eliminates the necessity to change the elliptic solver boundary conditions and can significantly reduce the time required to generate a usable volume grid, by reusing good potions of poor quality grids.

9.2 Smoothing with Trans-Finite Interpolation

Smoothing surface and volume grids utilizing computational or physical based interpolants as basis functions is one set of methods that can be used. Another method that is not as powerful but just as important is the use of trans-finite interpolation (TFI), with the following syntax:

```
tfi dimension domain {iterations=} xyz[...]
```

where,

dimension is the dimension of the TFI. This can be either `2d` or `3d`.

domain is the physical or computational domain to be used for the TFI interpolants. The possible values for the argument can be `physical` or `parametric`.

iterations= this is the number of iterations to be performed to optimize the interpolants in three-dimensional TFI; 16 is sufficient.

> **NOTE:** This argument is only necessary when three-dimensional TFI is performed.

xyz[...] is the region of a grid block to be regenerated.

Besides grid smoothing, the capabilities of the **tfi** command include surface and volume grid generation (see chapter/section 11.1).

To employ TFI properly, the defining bounds of a region to be regenerated needs to be well posed.[20] For example, the defining edges or faces, of a surface or volume, respectively can not have sharp changes in grid line character. Illustrated in Fig. 9.13 are a set of regions that were chosen to perform TFI for smoothing purposes. The sharp changes in the edges

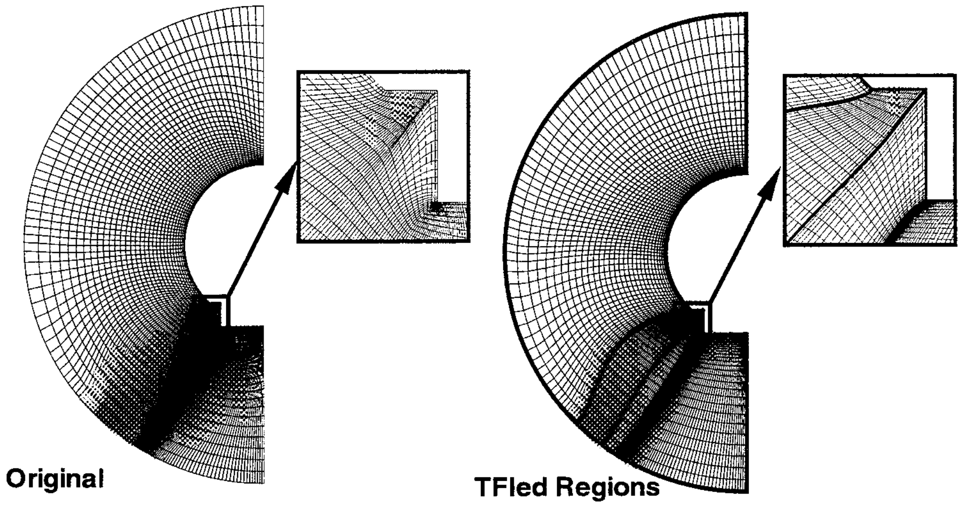

Figure 9.13: Poorly chosen boundaries used in the method of TFI for grid smoothing.

of the surface get propagated into the interior of the domain. The lack of orthogonality at the edges is also propagated onto the edges of the regions regenerated with TFI.

Conversely, applying the TFI regeneration to more appropriately chosen regions, the grid qualities at the boundaries that are good can be propagated onto the region. In the above example, different regions for performing TFI are chosen and a better grid results, as illustrated in Fig. 9.14. If good boundaries are available, TFI smoothing can be very robust.

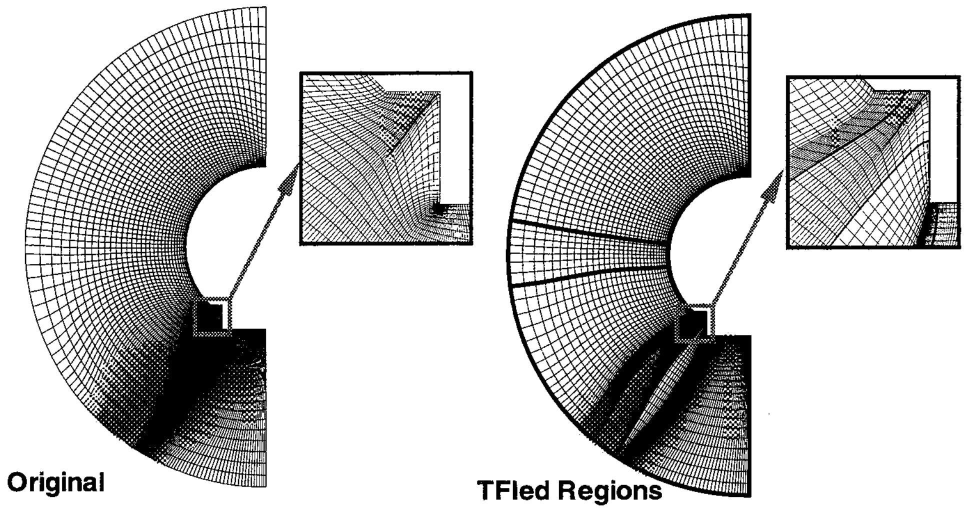

Figure 9.14: Appropriately chosen boundaries used in the method of TFI for grid smoothing.

But it can also do harm by not maintaining a smooth transition of cell sizes, as shown in Fig. 9.14. On the top of the configuration, the grid lines look much better. But there are spacing mismatches in the cross-sectional direction on the bottom. To help reduce the spacing mismatches, the parametric re-mapping may be needed. Although the TFI methods are robust and can offer a much easier alternative to parametric re-mapping, the latter may still be needed. Together, these methods create a powerful smoothing tool.

9.3 Vector Interpolation

Parametric re-mapping and TFI/LARCS methods do work well when combined with one another to do grid smoothing. But these approaches exclusively create smooth grids. The results will always be smooth, but other constraints such as clustering and grid line orthogonality may be compromised. Because parametric re-mapping and the TFI/LARCS methods are implicit, a more direct (i.e. explicit) method was developed and implemented into VGM.

The explicit method of doing grid smoothing that is in VGM is called hermite vector interpolation. The method is accessed through the **smooth** command, with the following syntax:

smooth *dimension direction definition_type order distribution_function xyz[...] boundary_condition*

where,

dimension is the dimension of the smoothing. As implemented, this can only be `1d`. The `2d` or `3d` capabilities will be added to VGM in the future for PDE solving.

direction is the direction to smooth the grid. The possible values are `I-direction`, `J-direction`, or `K-direction`.

definition_type determines whether the defining grid that describes a surface or volume retains the current shape or is permitted to change (similar to the subface parametric mode of GRIDGEN). The possible values are `fixed` or `general`.

NOTE: Only the `general` option is implemented.

order is the degree to which the derivatives are computed in a specified direction of smoothing. It is specified by `order=#` where #+1 points are used to construct the necessary derivatives.

distribution_function this is the function to be used for the smoothing. The functions possible are:

1. `equal`
2. `vinokur`[14](Δs_{begin},Δs_{end})
3. `cubic`(Δs_{begin},Δs_{end})
4. `vin2cub`(Δs_{begin},Δs_{end},ratio)
5. `sin`
6. `-sin`
7. `cos`
8. `laura`(Δs_{begin},S_{max},fstr,ep0,fsh)
9. `func`(*filename*) or func(*array_variable*)

NOTE: To retain the existing distribution or existing cell sizes for the distribution functions that require control parameters, include the argument *keepcurrent* with the **smooth** command.

xyz[...] is the region of a grid block to be smoothed.

boundary_conditions are specified as Dirichlet or fixed grid boundary conditions. The possible values are specified by `dirichlet=(...)` and the control parameters are dependent on the direction of smoothing; if smoothing is to be done in the I-direction, Dirichlet BC's are possible at `jmin`, `jmax`, `kmin`, and `kmax`.

NOTE: If one of the Dirichlet BC's is missing, that edge will have a Neumann boundary condition in which the grid will be smoothed at that edge.

The smooth command currently supports one dimensional smoothing. But this type of smoothing can do two and three dimensional smoothing, implicitly. The smoothing of grid lines is done by computing the derivatives at end points of the bounding region of the grid to be smoothed, in the direction of the smoothing. These derivatives are then used to interpolate from the beginning index to the ending index in the direction chosen. The derivatives determine the continuity and precision of smoothness that will result from the interpolation, as shown in Fig. 9.15. Typically, the second order derivatives are sufficient to

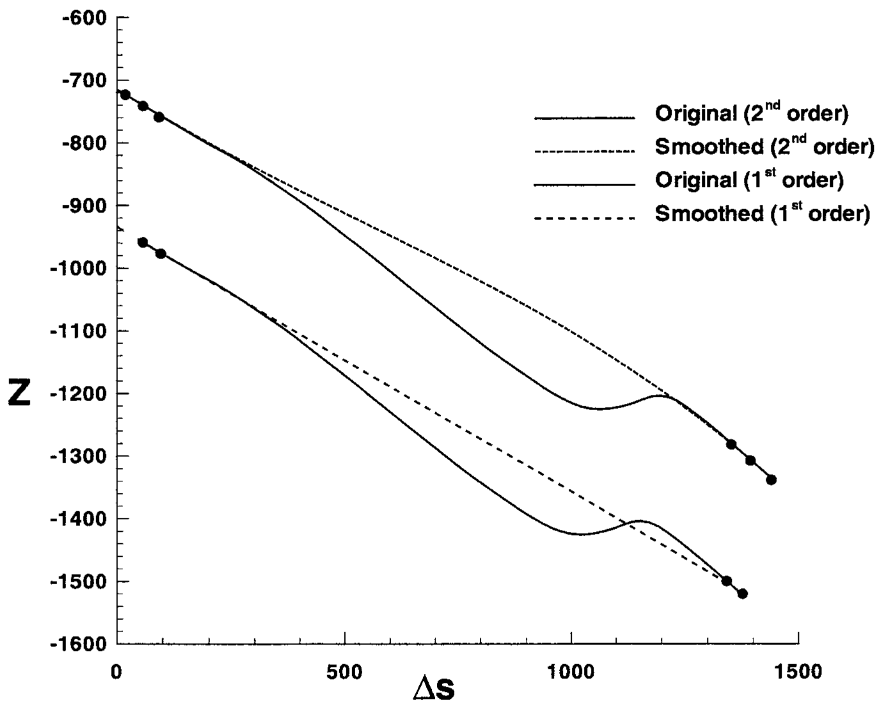

Figure 9.15: Order of derivatives for one dimensional smoothing.

produce the smoothest grids.

The one dimensional smoothing has boundary conditions that control the extent to which the region is smoothed at the boundaries. The method uses a blending of four grid points in the cross-directional (i.e. directions not including the direction of smoothing) indices from the region boundary to the interior. There are two type of boundary conditions that can be applied, Dirichlet and Neumann. As pertaining to the solution of PDE's, the Dirichlet boundary condition keeps a boundary fixed thereby using the four point blend to the interior, while the Neumann boundary condition allows the boundary to move to satisfy a derivative condition. For one dimensional smoothing, the Neumann derivative condition is slope continuity from the interior to the boundary edge or surface that is allowed to move from its original positions. For example, to smooth a surface with kinked grid lines, as illustrated in Fig. 9.16 the grid lines in the identified region between the darker grid lines needs smoothing. Clearly, smoothing in the I-direction would be best because the discontinuity occurs along

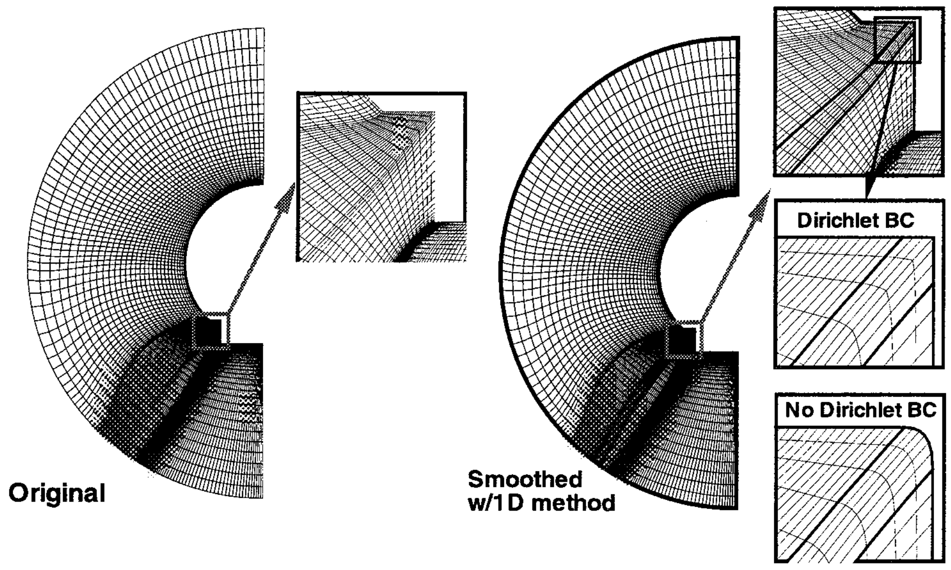

Figure 9.16: Identified region for one-dimensional smoothing.

the I-varying grid lines. To smooth these grid lines, the following commands can be used:

```
# Do not use Dirichlet boundary conditions:
smooth 1d i-direction general order=2 keepcurrent equal \
        xyz[1,1,...]  dirichlet=()

# Use Dirichlet boundary conditions:
smooth 1d i-direction general order=2 keepcurrent equal \
        xyz[1,1,...]  dirichlet=(jmin,jmax)
```

Script 9.8: 1D smoothing by vector interpolation with Neumann and Dirichlet boundary conditions.

The effect of this smoothing is illustrated on the right in Fig. 9.16. Notice that the non-Dirichlet boundary condition does not preserve the cross-directional boundary. This may be appropriate if the non-Dirichlet applied boundary does not match to another volume grid, or is not at the wall of a configuration.

Applying the one-dimensional smoothing in three dimensional space can provide a powerful mechanism to recover from parametric re-mapping or TFI type smoothing. As performed earlier, a blending of the sphere-cone grid in the `3dp` portion of the **blend** command for smoothing in the K-direction, can be done in the I direction as well. The I-direction smoothing may result in a better grid in the K-direction because that would be one of the dependent directions of the TFI. By blending in the K-direction, a decoupling of the redistribution dependency occurs and can produce crossed grid lines in the K-direction because of the non-dependency. Application of the `3dp` **blend** in the I-direction establishes the dependency in the J- and K-directions and can lead to a grid that may not have crossed grid lines in the I-direction, but could produce a grid that is kinked at the interface to the undisturbed grid. This kinked interface could be easily smoothed with 3DTFI and the **smooth** command, as performed by the following commands:

```
# Redistribute region of poor elliptic PDE boundary conditions:

allocate dsk1[45,363,33]

set dsk1 = dska(xyz[1,1,1-45])

blend dsk1[1-0:0,1-0:0] 3dp k-direction physical interpolation=tfi \
        xyz[1,1,1-45,:0,:0]
redist k-direction spline arclength physical points=33 func(dsk1) \
        xyz[1,1,1-45] newblock=no

# Improve continuity from corrected grid to original block:

tfi 3d arclength iterations=16 xyz[1,1,45-51]
smooth 1d i-direction general order=2 keepcurrent equal \
        xyz[1,1,43-47] dirichlet=(jmin,jmax,kmin,kmax)
```

Script 9.9: Single direction smoothing applied to 3D.

and illustrated by Fig. 9.17. As is evident by the smooth grid lines and the improved orthogonality at the symmetry planes on the sphere-cone grid, this blending and smoothing is an alternative to a K-directionally based `3dp` **blend**This also illustrates one of the strengths of the VGM language; there is more than one way to solve a problem, given the command structure and capability.

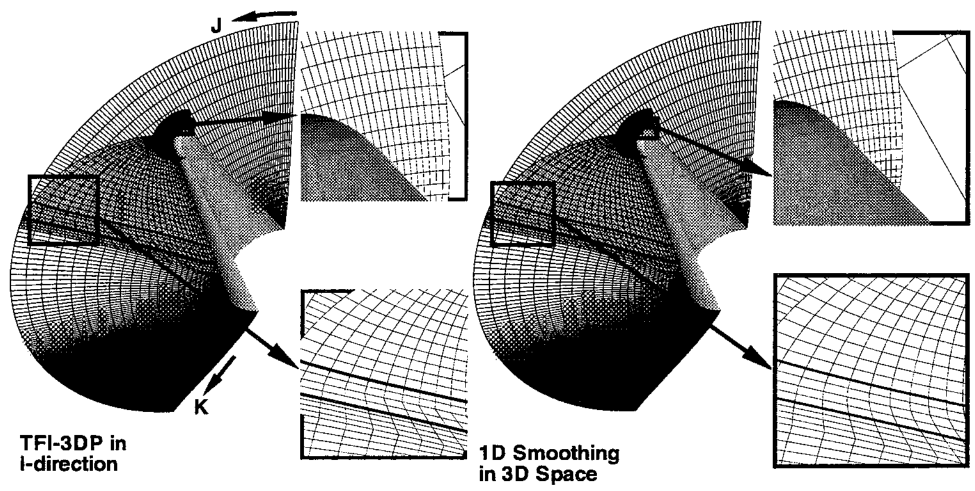

Figure 9.17: Three-dimensional smoothing of an improved grid to the original volume grid.

9.4 Summary of Techniques

As identified by the Parametric Re-mapping, TFI, and Hermite Vector Interpolation schemes presented, there are numerous ways to smooth surface and volume grids. The VGM language supports a variety of commands that when used in conjunction with one another form the basis for a powerful set of tools. Each command by itself is unique and simple, but when combined create tools of increased complexity. This flexibility, when used appropriately can enable the smoothing of poorly defined or otherwise generated surface and volume grids quickly and efficiently.

Chapter 10

CFD Adapting

10.1 Coarse Grid to Fine Grid Adaption

One of the main purposes of developing the **blend** command of section 9.1.3, was to implement a coarse grain to fine grain grid adaption technique for Computational Fluid Dynamic (CFD) simulations. CFD solutions are often started on coarse grids for computational efficiency, then interpolated to finer grids for the final analysis. Examples of fine and coarse grids are illustrated in Fig. 10.1. After the flow field is established, the coarse grid is sometimes adapted to some flow field characteristics such as pressure to resolve a shock wave. The solution is continued with the adapted grid until a relatively converged flow solution is completed. Then the coarse grid is used as a template for adapting the fine grid and the solution is interpolated onto the adapted fine grid for final analysis.

The grid adaption is accomplished by copying the arclength distributions from the body to the outer domain from the coarse grid to the fine grid. Then the **blend** command is used to interpolate with a `spline` blending function in the I-direction holding the copied K-lines constant and parametrically re-mapping the regions between the K-lines. Then a similar re-mapping of arclengths is done in the J-direction and an adapted grid results. The commands used to do this adaption are illustrated in script 10.1 and the results are shown in Fig. 10.2.

This adaption technique can also use the `3dp` **blend** in the body to outer domain direction after the distributions from the coarse grid are copied into the fine grid, as illustrated in script 10.2. This method may produce dependency problems in the K-direction (body to outer domain) and result in crossed grid lines, as explained in section 9.3 of chapter 9. But it is an alternate and may be necessary, depending on the extent of the adaption.

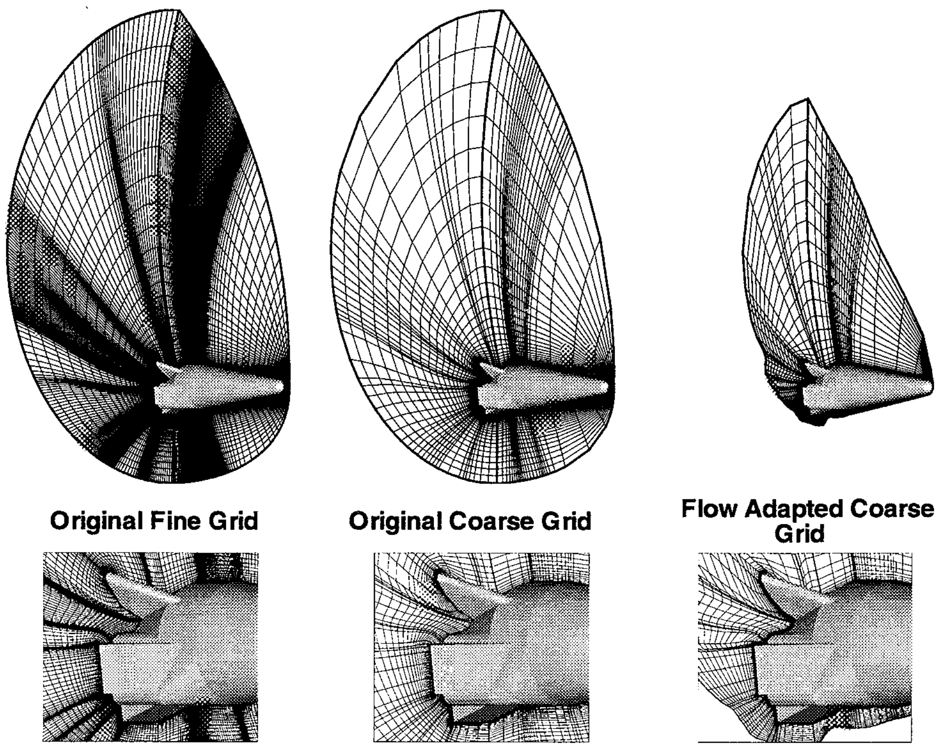

Figure 10.1: Initial coarse grid of a fine grid used on a McDonnell Douglas proposed X33 configuration.

```
copydist k-direction spline arclength xyz[1] xyz[2,1,:4,:4]
allocate dsk1[161,197,65]
set dsk1 = dska(xyz[2])
blend dsk1[:4,:4] 1d i-direction physical interpolation=spline \
        xyz[1,1,:4,:4]
blend dsk1[,:4] 1d j-direction physical interpolation=spline \
        xyz[1,1,,:4]
redist k-direction spline arclength physical points=65 func(dsk1) \
        xyz[2] newblock=no
```

Script 10.1: Coarse to fine grid adaption with 1D smoothing.

Both of the above methods can also be used to adapt or blend solution data from the coarse grid to the fine grid. As discussed in chapter 4 section 4.2.3, the flow field solutions to a grid can be loaded into VGM and each will be assigned a unique variable. This variable can be used in place of the grid blocks (`xyz[...]`) to do the same type of blending as performed on the grid. The only effect the blending has is on the dimensionality of the

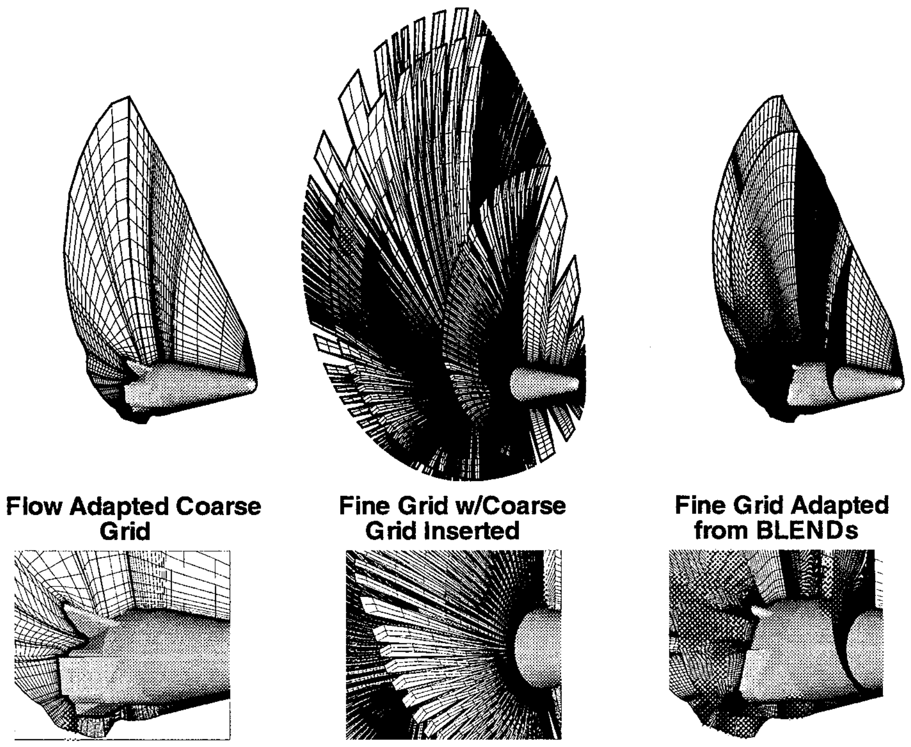
Flow Adapted Coarse Grid
Fine Grid w/Coarse Grid Inserted
Fine Grid Adapted from BLENDs

Figure 10.2: Fine adapted grid based on the CFD solution of a coarse grid.

```
copydist k-direction spline arclength xyz[1] xyz[2,1,:4,:4]
allocate dsk1[161,197,65]
set dsk1 = dska(xyz[2])
blend dsk1[:4,:4] 1d i-direction physical interpolation=spline \
       xyz[1,1,:4,:4]
blend dsk1[:4,:4] 1d j-direction physical interpolation=spline \
       xyz[1,1,:4,:4]
blend dsk1[:4,:4] 3dp k-direction physical interpolation=tfi \
       xyz[1,1,:4,:4]
redist k-direction spline arclength physical points=65 func(dsk1) \
       xyz[2] newblock=no

smooth 1d i-direction general order=2 keepcurrent equal \
       xyz[1,1,56-59] dirichlet=(jmin,jmax,kmin,kmax)
```

Script 10.2: Coarse to fine grid adaption with 3D planar smoothing.

solution and trends in the data may not reflect actual phenomenon. Assuming a density (ρ) for grid system 3 and block 2, the following commands can be used to interpolate a coarse grid solution to a fine grid:

```
#
# In lieu of copydist insert the coarse solution into the fine:
#

allocate rho_n3_b2_fine[161,197,65]

set rho_n3_b2_fine[:4,:4] = rho_n3_blk2

blend rho_n3_b2_fine[:4,:4] 1d i-direction physical interpolation=spline \
        xyz[1,1,:4,:4]
blend rho_n3_b2_fine[,:4] 1d j-direction physical interpolation=spline \
        xyz[1,1,,:4]
```

Script 10.3: Coarse to fine solution adaption using 1D smoothing.

NOTE: The **redist** is not needed because the function space is being manipulated not the grid.

Using the procedure, a coarse grid can be used as a template to adapt a fine grid and the respective flow solution, in preparation for a complete body CFD simulation.

10.2 Adaption by Changing Grid Densities

A second technique that can be used for adaption is the use of the **redist** command. If a volume grid is adapted with the shock alignment procedure within the LAURA code, the outer domain (i.e. bow shock) may need improved grid resolution. To improve the resolution, two separate regions limited by the body to outer domain index (K) can be created such that the bow shock is isolated at some point close to the outer domain, as illustrated in Fig. 10.3.

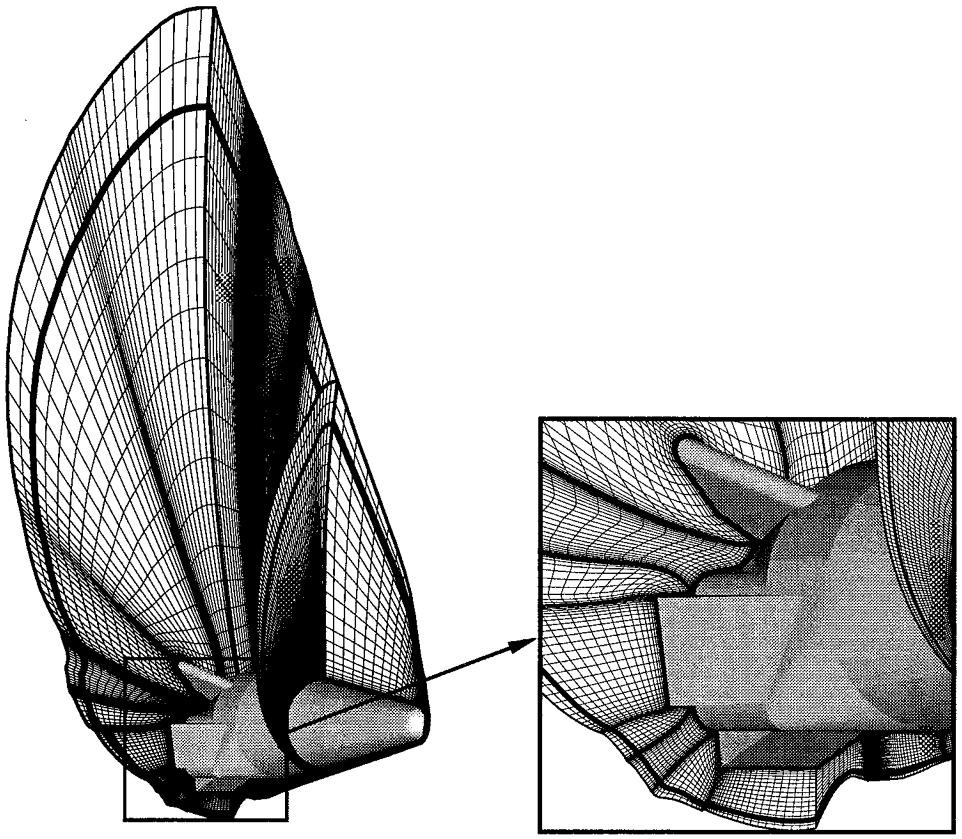

Figure 10.3: Initially adapted grid from ALGNSHK in the LAURA code.

Then these two regions can be adapted using the computational domain, keeping the wall cells at the current heights, and reducing the bow shock index cell points by 50%, using the following commands:

```
redist k-direction spline arclength parametric points=57 vinokur(1.,0.5) \
       xyz[1,1,,,1-61] newblock=yes
redist k-direction spline arclength parametric points=9 vinokur(0.5,1.)  \
       xyz[1,1,,,61-0] newblock=yes

# Merge the blocks:

allocate xyz[161,197,65]
set xyz[4,1,,,1-57] = xyz[2]
set xyz[4,1,,,57-0] = xyz[3]
```

Script 10.4: Grid adaption using varying grid densities.

and is illustrated in Fig. 10.4. Clearly this technique is fast and efficient and can provide

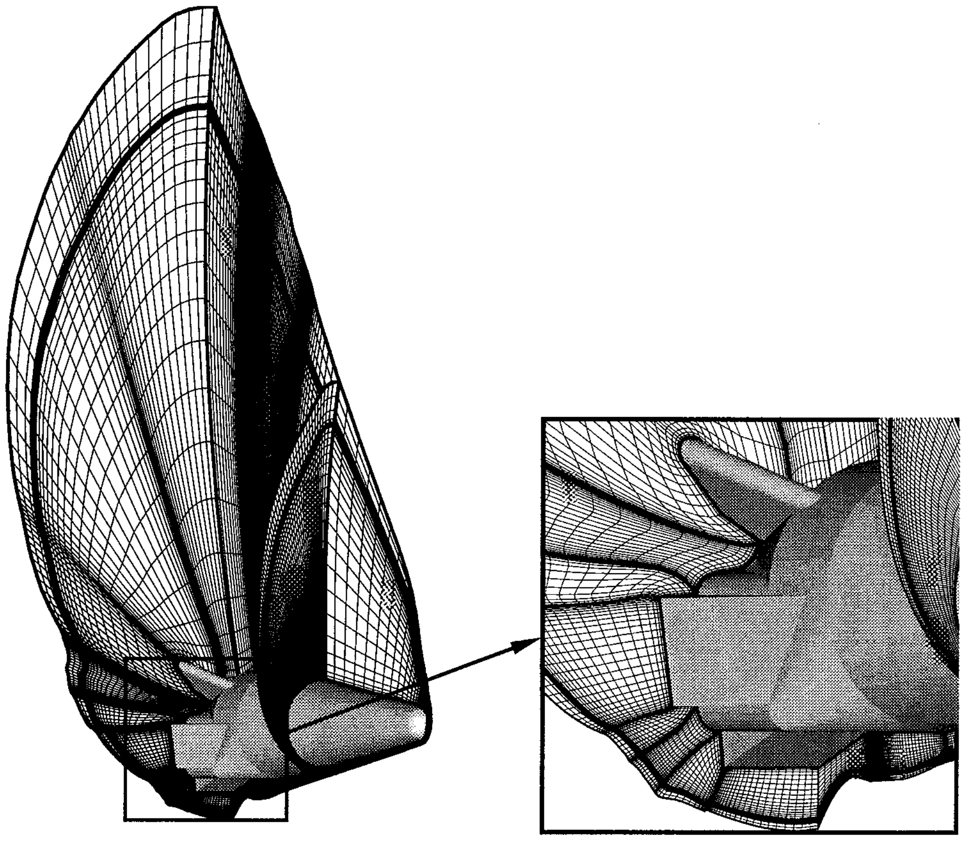

Figure 10.4: Re-adapted grid to improve bow shock capture and modeling.

improved grid resolution in the vicinity of the bow shock. This method can also be used in other directions to capture wing leading edge bow shock gradients, or any other gradients requiring improved grid resolution.

10.3 Summary of Adaption Techniques

Though the techniques of grid adaption are limited, each provides a tool not available by other software. The grid adaption techniques of VGM are also augmented by the rest of the manipulation language which can offer increased flexibility towards the application of grid adaption. Poor grid adaptions can be smoothed while new adaptions can be performed in a completely different manner. Either way, VGM provides a powerful set of tools to do grid adaption and can be very useful when used properly.

Chapter 11

Volume Generation

While VGM is inherently designed to do grid manipulations there are a few methods available for doing grid generation. These methods include:

- Three-Dimensional Trans-Finite Interpolation (3DTFI);
- 3DTFI coupled with intermediate definition planes;
- Straight line generation for grid repair;
- Coordinate corrections for planar assumptions.

Though the list is not very long, each offers an opportunity to keep the manipulations internal to VGM (items 1 and 3) or enables complex grid generation not available via standard 3DTFI methods (item 2). Each will be explained in further detail below.

11.1 Three-Dimensional Trans-Finite Interpolation

To generate a volume grid, the six defining faces of the volume need to be generated. Usually a code such as GRIDGEN is used to generate these faces. The volume is then generated by porting the faces to a volume grid generator and the code is executed to create the grid algebraically with 3DTFI. The porting and execution of a large grid generation such as GRIDGEN3D need not be done, for VGM can do the same generation.

The face definitions of the blocks that are to be generated, in the GRIDGEN format, are listed by K=constant faces, I=constant faces and J=constant faces, in the *.mlga file. To use these defining faces to develop a volume grid, each face has to be read in as a GRIDGEN surface and converted to the PLOT3D format by writing out the surface in that format. Since the J=constant faces have indices of K by I, these computational directions need to be switched on the output. The faces are then re-read, assigned to a grid block and the volume generated in VGM, using the following commands:

```
#   Step 1:  Load the GRIDGEN *.mlga file:
read example.mlga ascii gridgen multiple

#   Step 2:  Write out each face for each block, switching coordinates where
#        appropriate:
write blk1-face1-plt3d.g plot3d xyz[1,3] switch(x,y,z,k,i,j)
write blk1-face2-plt3d.g plot3d xyz[1,4] switch(x,y,z,k,i,j)
write blk1-face3-plt3d.g plot3d xyz[1,5] switch(x,y,z,j,k,i)
write blk1-face4-plt3d.g plot3d xyz[1,6] switch(x,y,z,j,k,i)
write blk1-face5-plt3d.g plot3d xyz[1,1]
write blk1-face6-plt3d.g plot3d xyz[1,2]

#   Step 3:  Read in the new faces:
read blk1-face1-plt3d.g
read blk1-face2-plt3d.g
read blk1-face3-plt3d.g
read blk1-face4-plt3d.g
read blk1-face5-plt3d.g
read blk1-face6-plt3d.g

#   Step 4:  Allocate a new block to hold the block:
allocate xyz[101,248,65]

#   Step 5:  Set the faces into the new block:
set xyz[8,1,1] = xyz[2,1]
set xyz[8,1,0] = xyz[3,1]
set xyz[8,1,,1] = xyz[4,1]
set xyz[8,1,,0] = xyz[5,1]
set xyz[8,1,,,1] = xyz[6,1]
set xyz[8,1,,,0] = xyz[7,1]

#   Step 6:  Perform 3DTFI on the new block:
tfi 3d arclength iterations=16 xyz[8,1]

#   Step 7:  Write out the new block:
write example1.vol plot3d xyz[8,1]

#   Step 8:  Done
quit
```

Script 11.1: Volume generation from GRIDGEN data.

Like any volume generated with 3DTFI, with faces that have discontinuities or encompass large volumes about configurations, the grid may not have grid-line characteristics that are suitable to CFD simulations. There may be a lack of near wall orthogonality or crossed grid lines creating negative volumes. To improve the quality of such grids, the volume can be subdivided into multiple blocks after the initial generation with 3DTFI and the interfaces smoothed with an elliptic PDE solver or VGM. Then these new interfaces can be imported into the original volume grid from which they came, and the volume grid updated with 3DTFI between the artificial boundaries created by the insertion of the interfaces. For example, the sphere-cone grid of the parametric re-mapping exercises is a truncated version of the full body sphere-cone-flare, illustrated in Fig. 11.1.

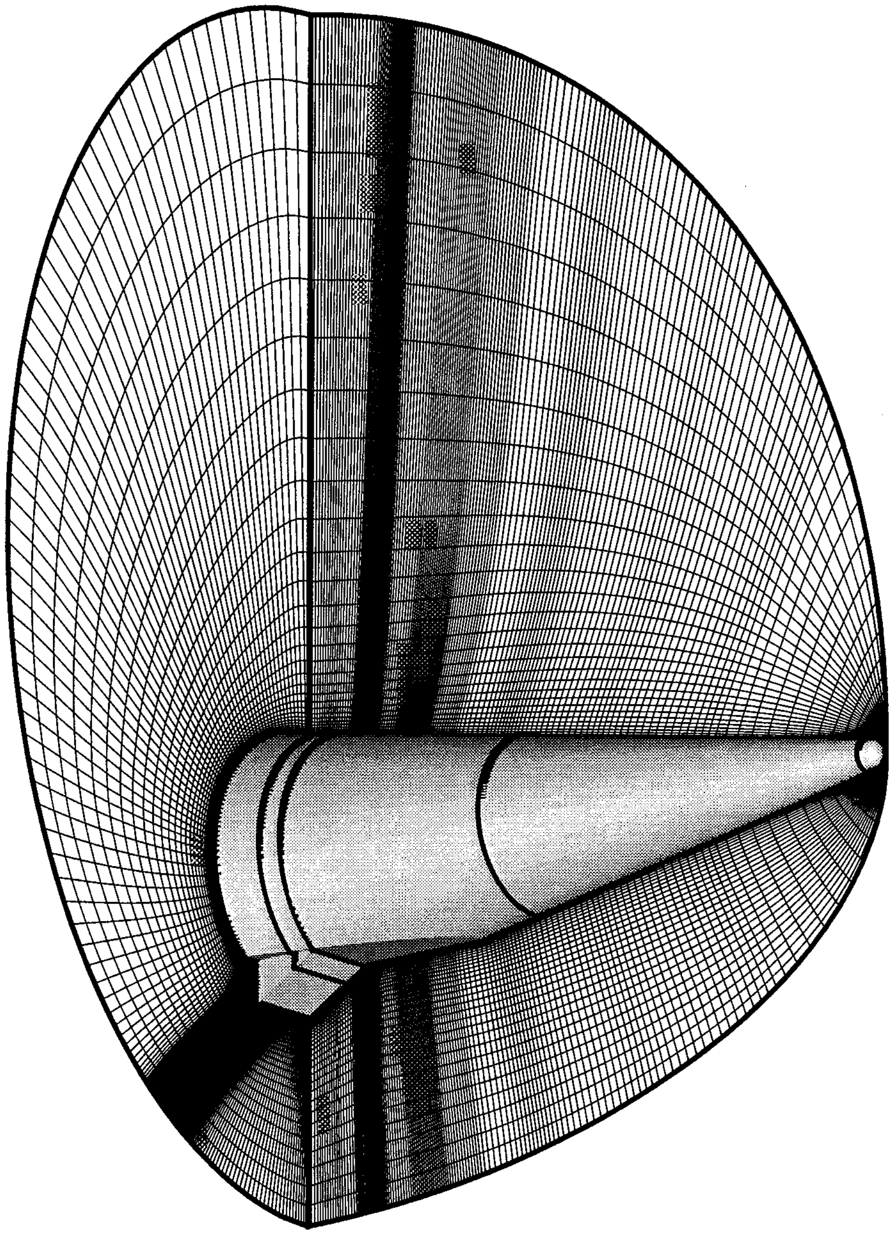

Figure 11.1: Full body sphere-cone-flare geometry.

For the purposes of the explanation of volume grid generation, the I-planes of 21, 101, 241 and 271 will be used to illustrate the effects of the grid being manipulated. These planes from the initial 3DTFI generation of the sphere-cone-flare geometry are illustrated in Fig. 11.2.

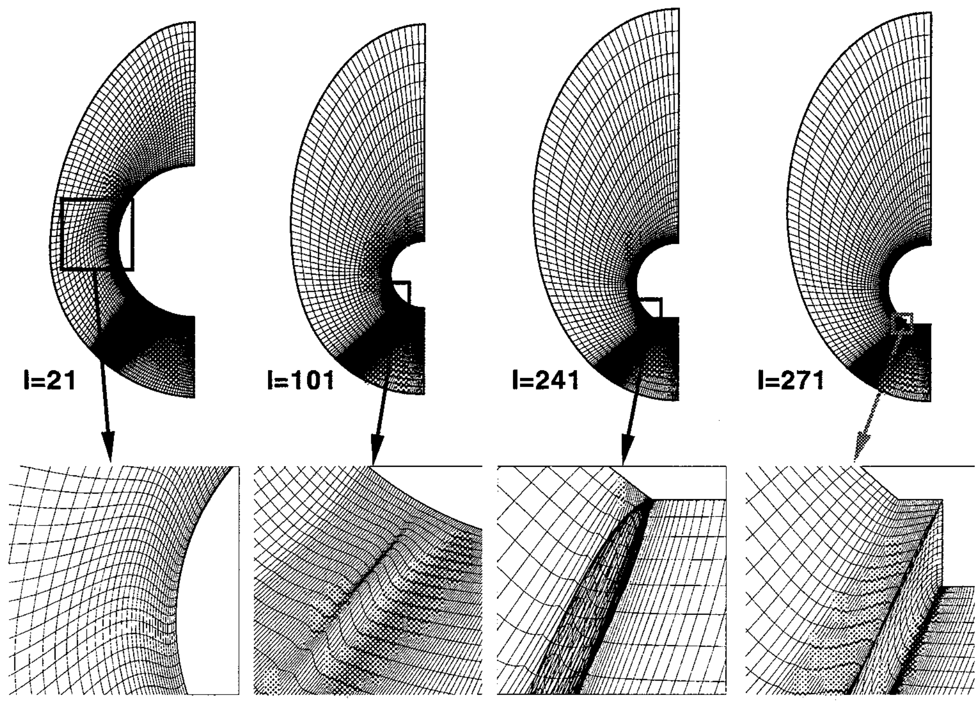

Figure 11.2: Full body sphere-cone-flare geometry initially generated with 3DTFI.

Close examination of this volume grid reveals that there are less than adequate grid line characteristics of orthogonality at the wall and negative volumes on the interior near the flapped portion of the flare. To improve this volume grid the planes of I=51, I=200 and I=255 were exported to GRIDGEN2D, smoothed and reinserted and the volume regenerated, using the VGM script above and script 11.2. The resulting grid is illustrated in Fig. 11.3.

Although the grid is by no means ready for CFD computing, the grid exhibits improved grid qualities because the crossed grid lines in I-plane of 101 are gone, and the orthogonality near the wall has improved. To further refine this volume grid, the 3DMAGGS code was employed to generate the nose portion and a flap region to establish better K-lines to use VGM smoothing techniques. Shown in Fig. 11.4 are the I-planes resulting from these elliptically generated blocks being inserted into the previous volume grid.

The final grid used for computations, after another set of grid manipulations to insert the nose and flap regions, as well as blend the new grids into the original volume grid, is illustrated in Fig. 11.5.

NOTE: The extra grid manipulations consisted of the sphere-cone `3dp` blending and a call to the **smooth** command to improve the grid line slope continuity from the 3DMAGGS generated grid to the original grid from which the zone originated.

```
#
#   Step 8:  (new) Read in the new interfaces and convert them
#           to PLOT3D format:
#
read xi51-new.grda gridgen ascii
read xi200-new.grda gridgen ascii
read xi255-new.grda gridgen ascii

write xi51-new.g plot3d xyz[9] switch(x,y,z,k,i,j)
write xi200-new.g plot3d xyz[10] switch(x,y,z,k,i,j)
write xi255-new.g plot3d xyz[11] switch(x,y,z,k,i,j)

read xi51-new.g
read xi200-new.g
read xi255-new.g
#
#   Step 9:  Insert the new interfaces:
#
set xyz[8,1,51] = xyz[12]
set xyz[8,1,200] = xyz[13]
set xyz[8,1,255] = xyz[14]
#
#   Step 10:  Regenerate the volume grid:
#
tfi 3d arclength iterations=16 xyz[8,1,1-51]
tfi 3d arclength iterations=16 xyz[8,1,51-200]
tfi 3d arclength iterations=16 xyz[8,1,200-255]
tfi 3d arclength iterations=16 xyz[8,1,255-0]
#
#   Step 11:  Write out the new block:
#
write example1-intermed.vol plot3d xyz[8,1]
#
#   Step 12:  Done
#
quit
```

Script 11.2: Zonal regeneration to augment volume generation.

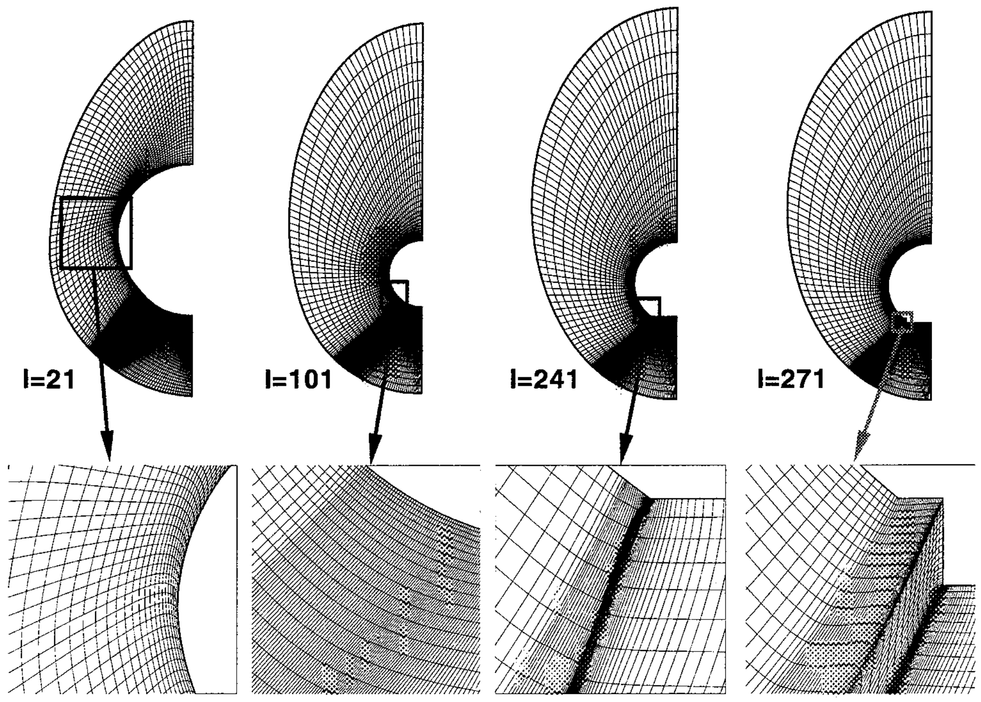

Figure 11.3: Full body sphere-cone-flare geometry, re-generated with 3DTFI and two intermediate fixed surfaces to control grid quality.

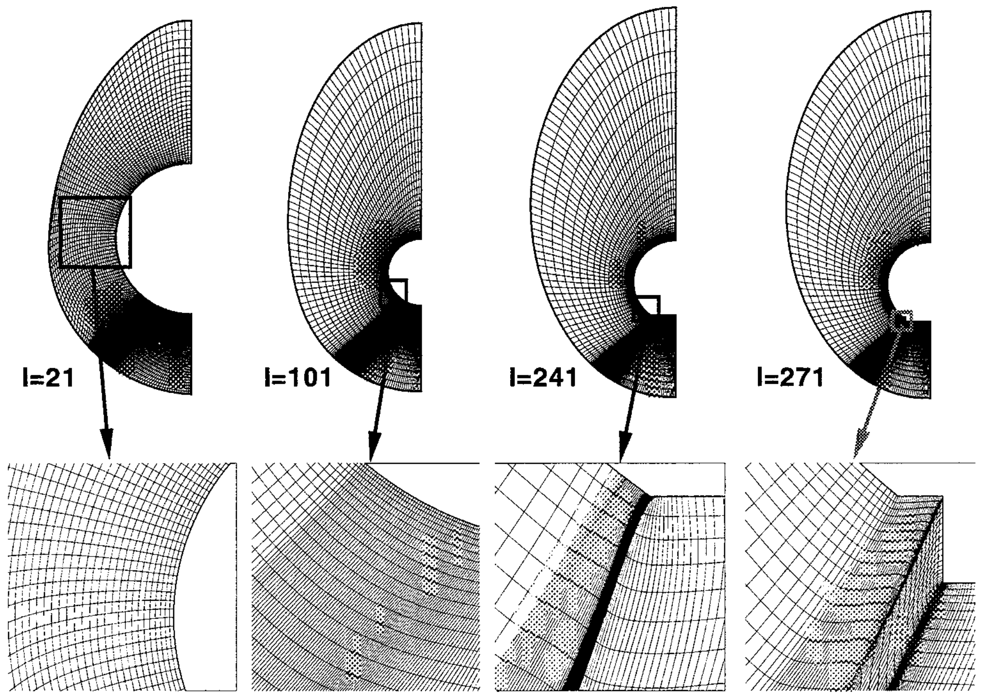

Figure 11.4: I-planes being tracked, after 3DMAGGS generation and VGM insertion of the nose and flap regions.

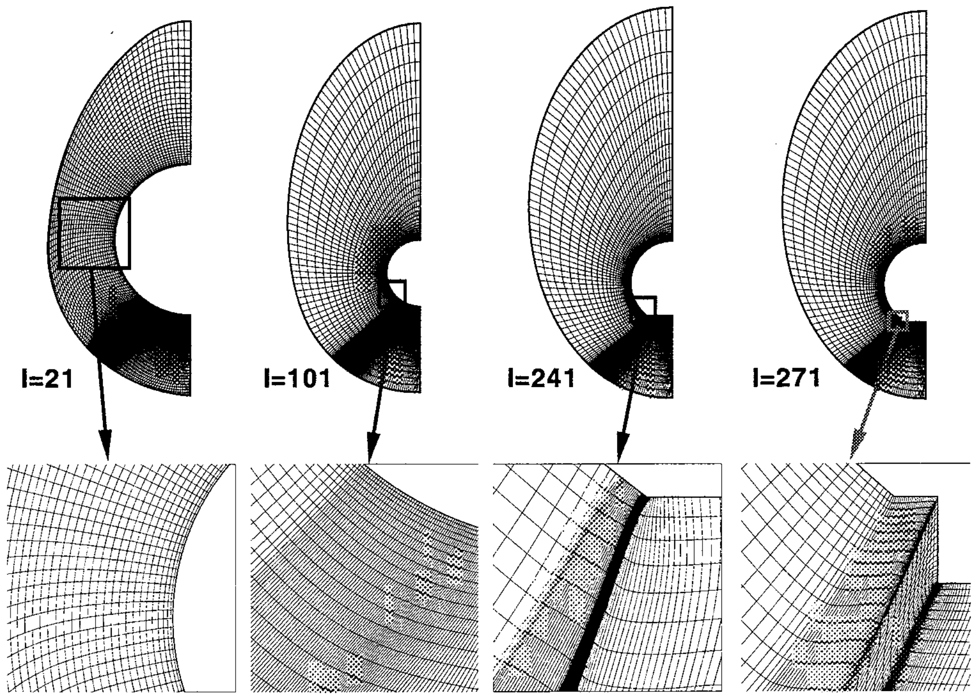

Figure 11.5: Final full body sphere-cone-flare geometry, re-generated with 3DMAGGS generated zones and VGM smoothing.

11.2 Correcting Grid Coordinates

In some cases of generating surface and volume grids, regions or entire faces of a surface or volume may be assumed to be flat or planar. During the evolution of a volume grid, these planar surfaces may be perturbed unintentionally. The VGM code can be used to correct these surfaces and force them to become planar, by using an extended set of grid intrinsics coupled with the **set** command. These extended grid intrinsics include:

- `x(xyz[...])` to extract the X-coordinate of a grid;
- `y(xyz[...])` to extract the Y-coordinate of a grid; and
- `z(xyz[...])` to extract the Z-coordinate of a grid.

The following VGM script is an example of how to use these intrinsics:

```
#
# Create core variables to extract a plane from a grid:
#

allocate xvar[161,1,65]
allocate yvar[161,1,65]
allocate zvar[161,1,65]

#
# Extract the coordinates with intrinsics:
#

set xvar = x(xyz[1,1,,1])
set yvar = y(xyz[1,1,,1])
set zvar = z(xyz[1,1,,1])
```

Script 11.3: Extraction of grid coordinates for manipulations.

These intrinsics are used in conjunction with the **set** command to extract existing grid data, converting the physical coordinates into internal variables. As internal variables they can be manipulated with the **blend** and **set** commands to do some re-generation and selective smoothing towards grid generation. For example, to force the Y-coordinate to be zero in the above example, thereby producing an X-Z symmetry plane, the following VGM script can be added:

```
set yvar = 0.0
set xyz[1,1,,1] = xvar yvar zvar
```

Script 11.4: Making an assumed plane, planar.

The last command above instructs VGM to set the physical coordinates of X, Y and Z for the appropriate grid locations to be the contents of variables xvar, yvar and zvar, respectively. Using the above construct to reset the physical coordinates of a volume grid, the internal variables are reconverted back to the grid data. One note of caution is due here; the left hand side of the last **set** command must have variables with the same physical limits, and can not be anything other than three internal variables in this construct (i.e. intrinsics are not allowed). Aside from this, the language construct for changing grid data and thereby re-evaluating or re-generating grid data can be done easily.

11.3 Straight Line Generation

During the adaption processes with some codes, a redistribution from VGM or the generation of a grid using the solution to hyperbolic PDE's, one of the 12 bounding edges of a block or 4 bounding edges of a surface may be corrupted. Such corruption usually requires regeneration of that edge using various line and curve types including ellipses, cubics, conics and in some cases straight lines. Though VGM is not designed to handle the former geometry types, the straight lines are possible through the use of the **redist** command. Utilizing the `linear` basis function construction argument of the **redist** command a line can be placed between two points, using any of the distribution functions explained in this manual. To generate a straight line, simply use the following command:

```
redist k-direction linear arclength physical points=9 vinokur(.1,1.)  \
      xyz[1,1,0,1,1-9:0] newblock=yes
set xyz[1,1,0,1,1-9] = xyz[2]
```

Script 11.5: Straight grid-line generation.

This set of commands will generate a straight line between points I=Imax, J=1 and K=1 to 9, add a `vinokur` distribution with an initial cell size of 0.1 and an ending cell size of 1.0

and place the new line into the original location from which it was extracted. This construct can be used to fix a volume grid, generated by HYPGEN, as illustrated in Fig. 11.6.

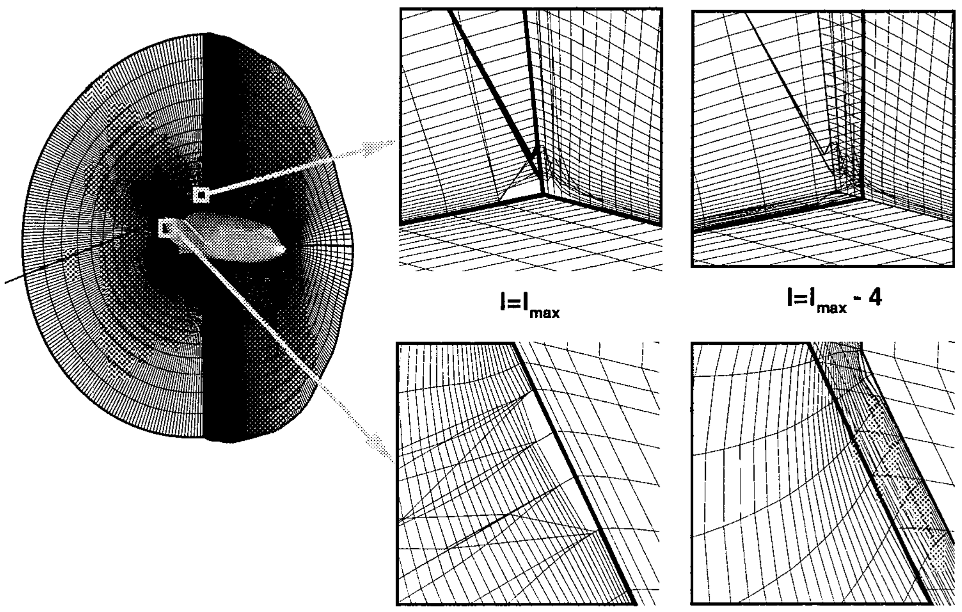

Figure 11.6: HYPGEN generated grid from a wall to an outer domain with a corrupted block edge.

To fix this grid, a straight line is placed between the K=1 and K=9 points at the exit of the volume grid on the top symmetry plane. Then the defining face grids are regenerated with 2DTFI and the volume updated with 3DTFI, using the following script:

```
redist k-direction linear arclength physical points=9 vinokur(.1,1.)  \
       xyz[1,1,0,1,1-9:0] newblock=yes
set xyz[1,1,0,1,1-9] = xyz[2]

#    Re-generate the block faces that use this edge:

tfi 2d arclength xyz[1,1,141-0,1,1-15]
tfi 2d arclength xyz[1,1,0,1-15,1-15]
tfi 3d arclength iterations=16 xyz[1,1,141-0,1-15,1-15]
```

Script 11.6: Correction of a block boundary to smooth a grid.

The resulting volume grid is illustrated in Fig. 11.7. As evident by the straight line inserted into the top back corner of the volume grid, this fix is more than adequate for CFD purposes

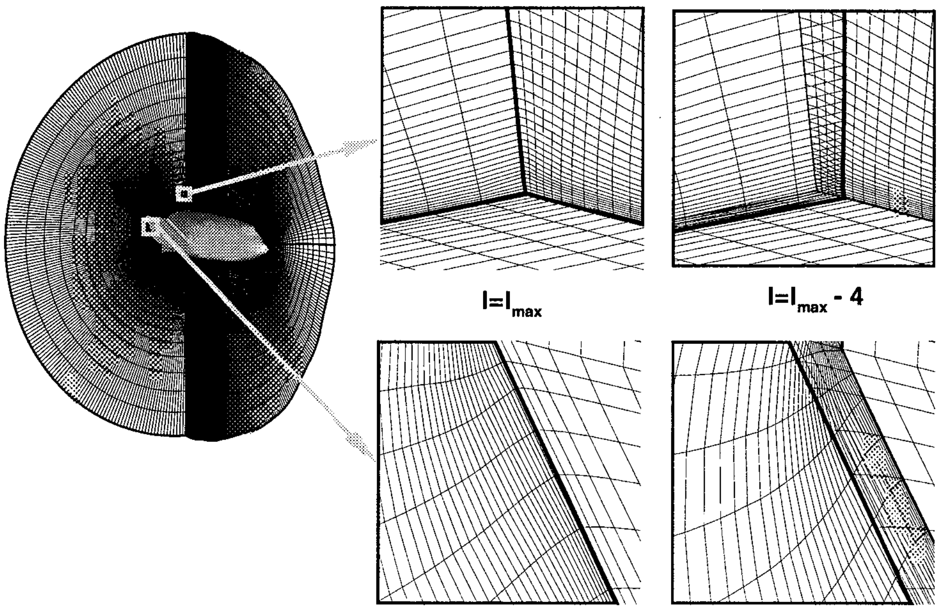

Figure 11.7: HYPGEN generated grid fixed with VGM using a straight line generation and other commands.

as the grid line orthogonality at the wall has been significantly improved and the volume grid freed of the corrupted defining block edge.

Also notice the trailing edge of the wing. A region about the negative volumes was regenerated simply by applying 2DTFI to the maximum I-plane and 3DTFI on a small region encompassing the fixed grid lines. These simple manipulations were added to the scripts modifying the top of the vehicle, thereby performing all necessary fixes at one time.

11.4 Summary of Generation Capabilities

Although VGM was never designed to do grid generation, the use of 3DTFI, the extended grid intrinsics and the generation of straight lines can provide an alternative to multiple codes used to generate or fix surface and volume grids. The techniques are simple, but once again, when combined with other VGM commands, create a powerful tool to manipulate existing grids.

Chapter 12

Tutorials

This chapter contains 6 individual simple operational tutorials, including:

1. Coarsening a volume grid;
2. Decomposing a single block volume grid into multiple blocks;
3. TFI grid smoothing;
4. Conversion of an inviscid grid to a viscous one;
5. Merging multiple blocks into a single block;
6. Combination of tutorials 3 and 4.

12.1 Tutorial I: Coarsening a Volume Grid

12.1.1 Purpose:

The purpose of this tutorial is to thin out or coarsen a dense volume grid to an ordered subset for CFD computing. The coarse grid will be used by CFD solvers to establish the true domain of the flow field, which is expected to lie within the limits of the existing grid. Upon adapting the grid to established flow field, this coarse grid can be used as a template to adapt the fine volume grid.

12.1.2 Steps To Be Used

To coarsen a volume grid, the following steps are used:

1. Read in the volume grid (/VGM/tutorials/ssv001f.g). The file is in PLOT3D, single block, Fortran Unformatted (binary) style, dimension and type, respectively.
2. Write out the coarse grid to ssv001f-coarse.g, by writing out every 4^{th} point in the I-direction, every other point in the J-direction and every 4^{th} point in the K-direction, using the LAURA code coordinate system.

These commands are coded into a script and executed either interactively or as a background process. Once the script has been executed, results must be viewed using appropriate graphics software such as TECPLOT^{TM} or FAST.

12.1.3 VGM Script

The script to perform this operation is:

```
#
#      This script coarsens a volume grid of unknown size:
#
#   Step 1:  Read in the volume grid into grid system 1.
#

read  /VGM/tutorials/ssv001f.g plot3d single binary

#
#   Step 2:  Write out every 4th I-point, every 2nd J-point and every 4th K-point:
#

write  /VGM/tutorials/ssv001f-coarse.g plot3d single binary xyz[1,1,:4,:2,:4]

#
#   Step 3:  Done; exit VGM.
#

quit
```

Script 12.1: Coarse grid generation from fine grid definition.

12.1.4 Results

The results of this script are illustrated in Fig. 12.1: Notice that the dimensions of the original grid are (161 X 129 X 33), and by using increments of 4, 2 and 4 for the I-, J-, and K-directions, respectively, the dimensions of the coarse grid are (41 X 65 X 9). The wall features are preserved by the coarse grid even though it contains significantly fewer points. This coarse grid can be used to reduce the time required to establish a CFD simulation on this vehicle.

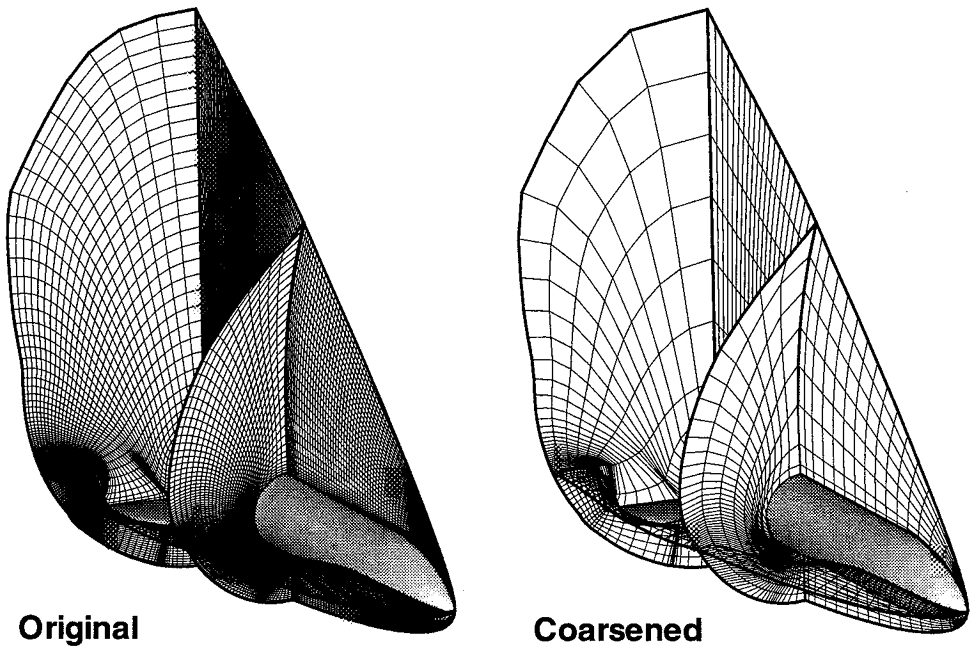

Figure 12.1: Results of thinning out a dense volume grid.

12.2 Tutorial II: Decomposing a Single Block Volume Grid

12.2.1 Purpose:

The purpose of this tutorial is to decompose a volume grid from a massive single block into a set of multiple blocks. By doing so the grid becomes more manageable and applicable to parallel processing.

12.2.2 Steps To Be Used

To decompose a single block into multiple blocks, the following steps are used:

1. Read in the volume grid (/VGM/tutorials/ssv001f.g). The file is in PLOT3D, single block, Fortran Unformatted (binary) style, dimension and type, respectively.

2. Allocate internal blocks for 4 individual grids and extract the volume grids represented by:

    - Block 1: I=1-81, J=1-129, K=1-33;
    - Block 2: I=81-161, J=1-69, K=1-33;
    - Block 3: I=81-161, J=69-87, K=1-33; and
    - Block 4: I=81-161, J=87-129, K=1-33

3. Combine the blocks into a new grid set.

4. Write out the new multiple block decomposition to ssv001f-decomp.g.

12.2.3 VGM Script

The script to do this tutorial should be:

```
#
#      This script decomposes a volume grid into 4 blocks:
#
#   Step 1:  Read the volume grid into grid system 1 (xyz[1]).
#
read  /VGM/tutorials/ssv001f.g plot3d single binary
#
#   Step 2:  Allocate internal grid blocks and decompose the domain:
#          Grid System 2, block 1 will contain new Block 1.
#          Grid System 3, block 1 will contain new Block 2.
#          Grid System 4, block 1 will contain new Block 3.
#          Grid System 5, block 1 will contain new Block 4.
#
allocate xyz[81,129,33]
allocate xyz[81,69,33]
allocate xyz[81,19,33]
allocate xyz[81,43,33]

set xyz[2] = xyz[1,1,1-81]
set xyz[3] = xyz[1,1,81-0,1-69]
set xyz[4] = xyz[1,1,81-0,69-87]
set xyz[5] = xyz[1,1,81-0,87-0]
#
#   Step 3:  Combine the new blocks into a set for writing the decomposition:
#
combine xyz[2] xyz[3] xyz[4] xyz[5]
#
#   Step 4:  Write the decomposed volume grid:
#
write  /VGM/tutorials/ssv001f-decomp.g plot3d multiple binary xyz[6]
#
#   Step 5:  Done; exit VGM.
#
quit
```

Script 12.2: Domain decomposition from a single block topology.

NOTE: Remember that the Grid System number always increments upon the allocation of a new grid block or combination of multiple blocks.

12.2.4 Results

The results of this script are illustrated in Fig. 12.2:

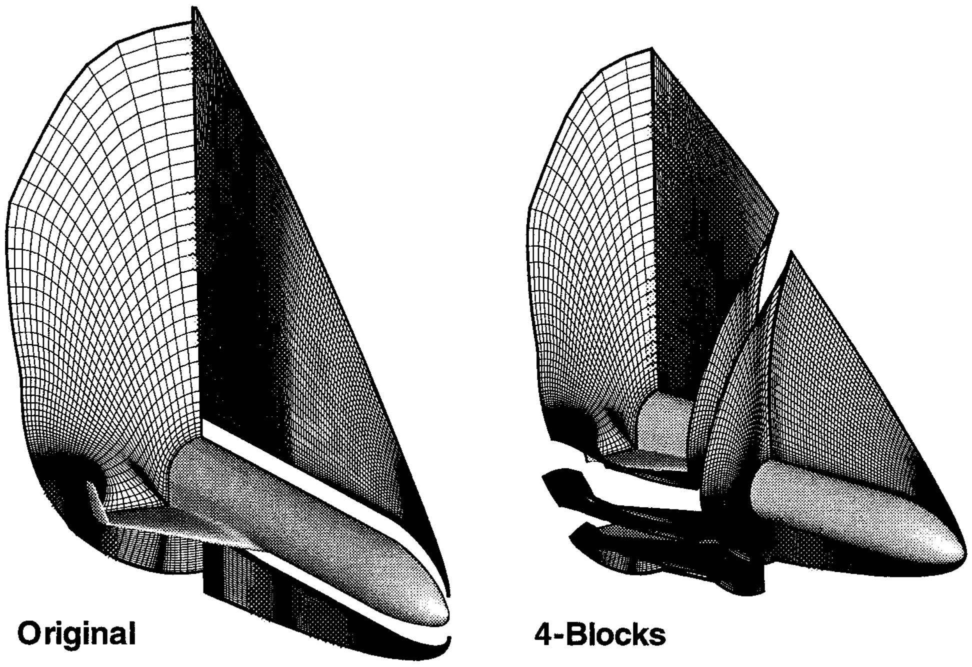

Figure 12.2: Results of decomposing a single block into multiple blocks.

12.3 Tutorial III: TFI Grid Smoothing

12.3.1 Purpose:

The purpose of this tutorial is to smooth a volume grid with either poor grid lines or negative volumes resulting from volume generation or grid adaption. In this example, parts of the interior grid and a portion of one of the grid boundaries (I=161) have been corrupted. Two dimensional TFI will be initially used to smooth the grid boundary the 3-D TFI will be used to repair the grid interior. The method, as detailed in chapter 9 section 9.2, offers a possible "quick fix" to problem regions.

12.3.2 Steps To Be Used

To smooth a volume grid using TFI, the following steps are used:

1. Read in the volume grid (/VGM/tutorials/ssv001f.g). The file is in PLOT3D, single block, Fortran Unformatted (binary) style, dimension and type, respectively.

2. Modify the regions of:

   - Zone 1: I=161, J=34-52, K=1-35; (Grid boundary smoothing using 2-D TFI)
   - Zone 1: I=141-161, J=34-52, K=1-35; (3-D TFI)
   - Zone 2: I=81-161, J=104-129, K=1-33; (3-D TFI) and
   - Zone 3: I=106-115, J=36-98, K=1-23. (3-D TFI)

3. Write out the new smoothed volume grid ssv001f-tfi.g.

12.3.3 VGM Script

The script to do this tutorial should be:

```
#
#       This script smoothes a volume grid using TFI:
#
#   Step 1:  Read in the volume grid
read  /VGM/tutorials/ssv001f.g plot3d single binary
#
#   Step 2a:  Smooth Zone 1 (I=161, J=34-52, K=1-15):
tfi 2d arclength xyz[1,1,0,34-52,1-15]
#
#   Step 2b:  Smooth Zone 1 (I=141-161, J=34-52, K=1-15):
tfi 3d arclength iterations=16 xyz[1,1,141-0,34-52,1-15]
#
#   Step 2c:  Smooth Zone 2 (I=81-161, J=104-129, K=1-33):
tfi 3d arclength iterations=16 xyz[1,1,81-0,104-0]
#
#   Step 2d:  Smooth Zone 3 (I=106-115, J=36-98, K=1-23):
tfi 3d arclength iterations=16 xyz[1,1,106-115,36-98,1-23]
#
#   Step 3:  Write the smoothed volume grid:
write  /VGM/tutorials/ssv001f-tfi.g xyz[1]
#
#   Step 4:  Done; exit VGM.
quit
```

Script 12.3: Grid smoothing using various TFI dimensions.

12.3.4 Results

The results of this script are illustrated in Fig. 12.3:

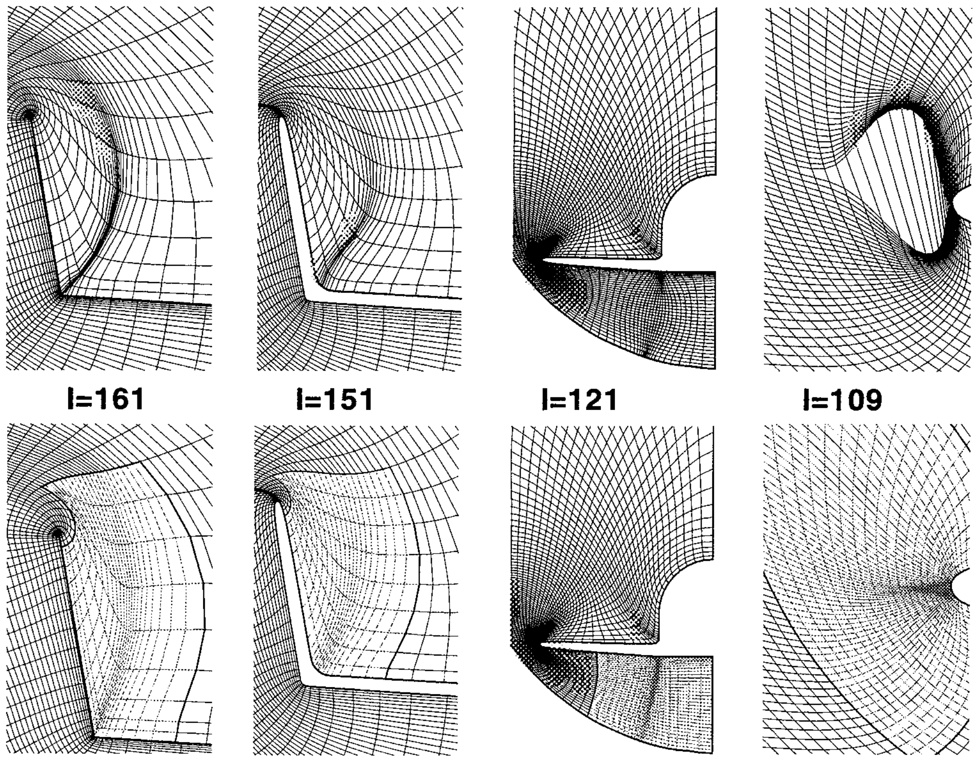

Figure 12.3: Results of smoothing a volume grid with TFI.

This script performs many operations, and offers a glimpse into what can be done in the VGM frame work. First, TFI is not limited to two or three-dimensions, only; both can be done in a single script as is typically the case. Second, to smooth a zone that abuts to a boundary, may require the generation of that boundary with TFI or other forms of smoothing, as done in zone 1. If the boundary is better than the volume grid, it is held fixed while the zone or volume is re-generated, as done in zone 2. Third, just because a zone has been re-generated with TFI does not mean that more precise zones internally cannot be re-generated. This offers the flexibility of establishing good boundaries and still providing enhanced smoothing, as done in zone 3.

12.4 Tutorial IV: Conversion of an Inviscid Grid to a Viscous Grid

12.4.1 Purpose:

The purpose of this tutorial is to convert a grid used for inviscid computations into one that is suitable for viscous computations. Inviscid grids typically have fewer points than a viscous grid because the shear and viscous effects are not being modeled. By comparison, to do a viscous computation, more points are needed to resolve the viscous effects including the packing of points near the wall to capture gradients in a boundary layer. To convert an inviscid grid to a viscous grid, the grid dimensions usually have to be increased in the direction of the strongest gradients, typically from the body to the outer boundary, and the spacing at the wall must be reduced.

12.4.2 Steps To Be Used

To convert an inviscid grid into a viscous one, the following steps are used:

1. Read in the volume grid (/VGM/tutorials/ssv001f.g). The file is in PLOT3D, single block, Fortran Unformatted (binary) style, dimension and type, respectively.

2. Increase the body to outer domain dimension (K) from 33 to 65 points and cluster points towards the wall (K=1) to capture boundary layer gradients.

3. Write out the new multiple block decomposition to ssv001f-decomp.g.

12.4.3 VGM Script

The script to do this tutorial should be:

```
#
#     This script converts an inviscid grid into a viscous one:
#
#   Step 1:  Read in the volume grid
read  /VGM/tutorials/ssv001f.g plot3d single binary
#
#   Step 2:  Increase the K-dimension and cluster points near the wall at K=1:
redist k-direction spline arclength parametric points=65 vinokur(.1,1.)\
          xyz[1] newblock=yes
#
#   Step 3:  Write the new volume grid:
write  /VGM/tutorials/ssv001f-redist.g plot3d single binary xyz[2]
#
#   Step 5:  Done; exit VGM.
quit
```

Script 12.4: Conversion of inviscid grid to viscous grid.

12.4.4 Results

The results of this script are illustrated in Fig. 12.4:

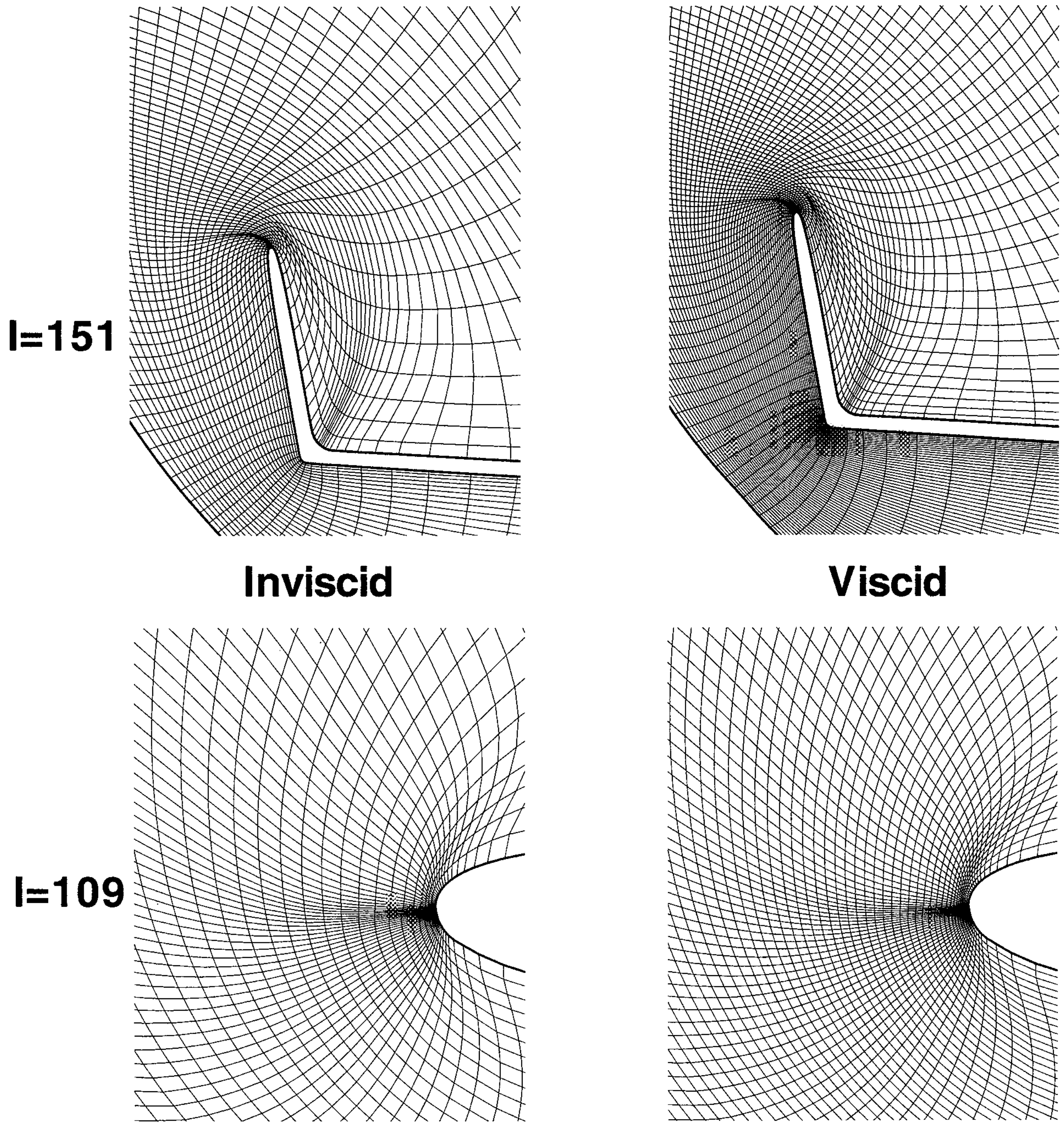

Figure 12.4: Results of converting an inviscid grid to a viscous one.

As explained in section 8.1, the `parametric` domain of the redistribution command enables the modifying of the number of grid points in one direction, while maintaining the overall grid line character of the source volume grid. This capability is exploited here, as well as the capability to scale the cell sizes at a boundary using the `vinokur` function with scaling percentages as opposed to physical cell sizes. In this tutorial, the cell sizes at the wall were scaled down to 10% of their current values, while the outer boundary cells were kept at their current values. Furthermore, by increasing the number of points from 33 to 65, the density of grid points from the wall to the outer boundary varies more smoothly than if the number of points were kept constant. Use of this technique can reduce the time to generate viscous grids for computations about complex geometries. Initiating a viscous flow solution with inviscid computations which are conservative (i.e. Euler computations which place the outer boundary farther from the wall than required), can ensure proper flow capture, an assumption of numerous CFD algorithms.

12.5 Tutorial V: Merging Multiple Block Decompositions Into a Single Block Volume Grid

12.5.1 Purpose:

The purpose of this tutorial is to convert the decomposition of a volume grid from multiple blocks into a single block. This may be advantageous for the manipulation of volume grid decompositions that have too many blocks to be tracked, or the reducing of the total number of blocks for a computation.

12.5.2 Steps To Be Used

To convert a multiple block volume grid into a single block, the following steps are used:

1. Read in the volume grid (/VGM/tutorials/ssv001f-decomp.g) from tutorial 2. The file is in PLOT3D, multiple block, Fortran Unformatted (binary) style, dimension and type, respectively.

2. Allocate one large volume grid to contain the 4 individual grids, and place them accordingly.

3. Write out the new single block decomposition to ssv001f-merge.g.

12.5.3 VGM Script

The script to do this tutorial is illustrated in script 12.5.

12.5.4 Results

Since the resulting volume grid is the same as in Tutorial 2, see Fig. 12.2.

```
#
#      This script converts a multiple block decomposition into a single block:
#
#   Step 1:  Read in the volume grid of Tutorial 2:

read  /VGM/tutorials/ssv001f-decomp.g multiple

#
#   Step 2:  Allocate internal grid block to contain all 4 blocks:
#

allocate xyz[161,129,33]

set xyz[2,1,1-81] = xyz[1,1]
set xyz[2,1,81-0,1-69] = xyz[1,2]
set xyz[2,1,81-0,69-87] = xyz[1,3]
set xyz[2,1,81-0,87-0] = xyz[1,4]

#
#   Step 3:  Write the single block volume grid:
#

write  /VGM/tutorials/ssv001f-merge.g plot3d single binary xyz[2]

#
#   Step 4:  Done; exit VGM.
#
quit
```

Script 12.5: Conversion of multiple block decompositions to a single block.

Chapter 13

Command Index

This chapter provides a comprehensive listing of all VGM commands and their individual syntaxes. This should be used for a reference only; the explanation of all the commands are in the previous chapters.

13.1 Input and Output

$\boxed{\textbf{read}\ \textit{filename}\ \{\textit{type}\}\ \{\textit{style}\}\ \{\textit{format}\}\ \{\textit{dimension}\}}$

where,

filename is the file name of the data to be **read**. The file name rules are as follows:

1. Limited to 60 characters in length;
2. Can not be identical to **read** arguments;
3. Can not contain ['s,]'s, \'s, or commas;
4. Are case sensitive;
5. May contain directory placement characters (./, ../ and ~)

format is the data format, `ascii`, `unformatted`, or `binary`. <default=unformatted>

style is the style the file is in; `gridgen`, `plot3d`, `laura` or `tecplot`$^{\text{TM}}$. <default=plot3d>

type is the type of data in the file; gridonly, solution(*ngsys*), or curve. <default=gridonly>

NOTE: The solution(*ngsys*) option requires a grid system number to attach the data to, to ensure there is one value for each grid point in each block. The variables loaded in this manner will have variable names of the form:

$\boxed{\textit{varname}\texttt{\_nNN\_blkBBB}}$

where the NN represents the Grid System number and the BBB represents the block number.

dimension represents the number of blocks in the grid set, `single` or `multiple`.
<default=single>

```
write filename {type} {style} {format} {dimension} xyz[...]  {orientation}
```

where,

filename is the file name of the data to be **written**. The file name rules are identical to the **read**command.

format is the data format, `ascii`, `unformatted`, or `binary`. <default=unformatted>

style is the style the file is in; `gridgen`, `plot3d`, `laura`, or `tecplot`(*variables*).
<default=plot3d>

> **NOTE:** The variables specifiable in the tecplot(*variables*) option include the physical coordinates (X, Y, and Z), the computational coordinates (I, J, and K), and *array* and *constant* variables in the form:

```
tecplot(x,y,z,i,j,k,dsj1)
```

type is the type of data in the file; gridonly, solution(*ngsys*), or curve. <default=gridonly>

> **NOTE:** The solution(*variables*) option requires a set of variables, similar to the `tecplot(...)` argument.

dimension represents the number of blocks in the grid set, `single` or `multiple`.
<default=single>

xyz[...] is the block or region or set of blocks to be written as a data set.

> **NOTE:** The block limits may be used in this command to select a range.

orientation is the physical and computational orientation of the grid. It is specified with the following argument:

```
switch(x,y,z,i,j,k)
```

where the physical coordinates are specified in the order to be written, and same with the computational coordinates.

> **NOTE:** The *orientation* basically changes the entire reference frame of the grid written. Beware, no check is done to determine if a left handed coordinate system is written.

13.2 Distributions

copydist *interpolant basis direction xyz1[...] xyz2[...]*

where,

interpolant is the parameterization to be used for the copy. The possible values for the argument are `arclength` or `normarc`.

basis is the interpolation basis to be used. The possible values can be `linear` or `spline`.

direction is the direction to copy the grid-point distributions, one for each computational index. The possible values are `I-direction`, `J-direction`, or `K-direction`.

xyz1[...] is the source grid block to get the grid-point distributions.

xyz2[...] is the destination grid block containing the grid-lines to be modified.

redist *domain basis direction interpolants points=# distribution_function newblock= xyz[...]*

where,

domain is the physical or computational domain to be used for the redistribution. The possible values for the argument can be `physical` or `parametric`.

basis is the interpolation basis to be used. The possible values can be `linear` or `spline`.

direction is the direction to redistribute the grid-point distributions, one for each computational index. The possible values are `I-direction`, `J-direction`, or `K-direction`.

interpolants is the type of parameterization to be used. The possible values are `arclength` and `normarc` (i.e. normalized arclength).

points=# is the number of points to be generated as a result of the redistribution.

distribution_function this is the function to be used for the redistribution. The functions possible are:

1. `equal`
2. `vinokur`[14](Δs_{begin},Δs_{end})
3. `cubic`(Δs_{begin},Δs_{end})
4. `vin2cub`(Δs_{begin},Δs_{end},ratio)
5. `sin`
6. `-sin`
7. `cos`

8. `laura`(Δs_{begin},S_{max},fstr,ep0,fsh)
9. `func`(*filename*) or func(*array_variable*)

newblock= specifies if the results of the redistribution are to be stored in a new grid system and grid block. Possible values are `yes` or `no`.

xyz[...] is the region of a grid block to be redistributed distributions.

13.3 Variable Manipulation

allocate *varname[I-limit,J-limit,K-limit]*

where,

varname is the array variable name to store computed grid parameters. The array variable name rules are as follows:

1. Limited to 60 characters in length;
2. Can not be identical to grid parameters (intrinsics);
3. Can not contain ['s,]'s, \'s, or commas;
4. Are not case sensitive;
5. Computational limits may not exceed the grid-point limits of VGM.

I-limits First Computational Index Limit

J-limits Second Computational Index Limit

K-limits Third Computational Index Limit

blend *varname[I-limit,J-limit,K-limit] direction dimension domain interpolation=* {*xyz[...]*}

where,

varname is the core variable containing an arclength parameter to be blended for smoothing a grid.

direction is the direction to blend the arclength parameters. The possible values are `I-direction`, `J-direction`, or `K-direction`.

dimension is the dimension of the blend. This can be either:

1. `1d` - single dimension
2. `2d` - two dimensions
3. `3dp` - two dimensions but planar by stepping through the third dimension
4. `3dw` - three dimensions

domain is the physical or computational domain to be used for the blending. The possible values for the argument can be `physical` or `parametric`.

NOTE: If the `parametric` domain is used, the parameterization of the domain being **blend**ed is based on the computational coordinates, and the `xyz[...]` need not be specified. Conversely, if the `physical` domain is to be used, the `xyz[...]` argument *must* be present and *must* have the same dimensions of the variable being **blend**ed.

interpolation= this is the interpolation scheme to be used to blend from one know index to another. The schemes possible are:

1. `linear`
2. `elliptic`[18]
3. `spline`
4. `tfi`
5. `larcs(#,#,#)`[19]

   NOTE: The `linear`, `elliptic`, and `spline` interpolation schemes are only available in one dimensional interpolation; the last two are for `2d`, `3dp` and `3dw` interpolation.

xyz[...] is the region of a grid block to be used for computing arclength blending functions if the *domain* is `physical`.

set *varname1[I-limit,J-limit,K-limit] = varname2[I-limit,J-limit,K-limit]*

-or-

set *varname1[I-limit,J-limit,K-limit] = ds*(xyz[ngsys,nblk,I-limit,J-limit,K-limit])*

where,

varname1 is the destination array variable name to store computed grid parameters or other variables.

varname2 is the source array variable or intrinsic (ds*) to be equated or computed, respectively. The only rule that must be followed is the computational region of each variable or intrinsic in the equate must be the same.

I-limits First Computational Index Limit of region to be **set**

J-limits Second Computational Index Limit of region to be **set**

K-limits Third Computational Index Limit of region to be **set**

`dsia` =	Arclength function in I-direction
`dsja` =	Arclength function in J-direction
`dska` =	Arclength function in K-direction
`dsin` =	Normalized arclength function in I-direction
`dsjn` =	Normalized arclength function in J-direction
`dskn` =	Normalized arclength function in K-direction

13.4 Block Manipulators

$\boxed{\textbf{allocate}\ \textit{xyz[I-limit,J-limit,K-limit]}}$

where,

xyz is a volume grid block, using the standard data structure.

I-limits First Computational Index Limit

J-limits Second Computational Index Limit

K-limits Third Computational Index Limit

NOTE: The gridsystem number and block number are not included in the **allocat**ion of the new grid block; only the computational limits are required. Also, this command will cause the grid system maximum to increase by 1 each time it is used.

$\boxed{\textbf{combine}\ \textit{xyz[ngsys1,nblk1] xyz[ngsys2,nblk2] ...}}$

where,

xyz is a source grid block. Subsequent *xyz*'s are other blocks to be added. There are some rules that can be used to govern which grid blocks are used:

1. `xyz[ngsys]` will get all the blocks in grid system `ngsys`
2. `xyz[ngsys,nblk_begin-nblk_end:nblk_increment]` will get those blocks that are referenced in the range from nblk_begin to nblk_end by nblk_increment

NOTE: This command will cause the increasing of the grid system maximum by 1 each time it is used.

$\boxed{\textbf{set}\ \textit{xyz[ngsys,nblk,I-limit,J-limit,K-limit]} = \textit{xyz[ngsys,nblk,I-limit,J-limit,K-limit]}}$

-or-

$\boxed{\textbf{set}\ \textit{xyz[ngsys,nblk,I-limit,J-limit,K-limit]} = \textit{x-variable y-variable z-variable}}$

where,

xyz on the left hand side is the destination block to store the results from extracting or merging grid blocks.

xyz on the right hand side is the source block to be extracted or merged. The only rule that must be followed is the computational region of each grid block in the equate must be the same.

I-limits First Computational Index Limit of region to be **set**

J-limits Second Computational Index Limit of region to be **set**

K-limits Third Computational Index Limit of region to be **set**

x-variable is the internal variable containing the X-coordinate of the block to be **set**

y-variable is the internal variable containing the Y-coordinate of the block to be **set**

z-variable is the internal variable containing the Z-coordinate of the block to be **set**

13.5 Grid Generation

smooth *dimension direction definition_type order distribution_function xyz[...] boundary_condition*

where,

dimension is the dimension of the smoothing. As implemented, this can only be `1d`. The `2d` or `3d` capabilities will be added to VGM in the future for PDE solving.

direction is the direction to smooth the grid. The possible values are `I-direction`, `J-direction`, or `K-direction`.

definition_type determines whether the defining grid that describes a surface or volume retains the current shape or is permitted to change (similar to the subface parametric mode of GRIDGEN). The possible values are `fixed` or `general`.

NOTE: Only the `general` option is implemented.

order is the degree to which the derivatives are computed in a specified direction of smoothing. It is specified by `order=#` where #+1 points are used to construct the necessary derivatives.

distribution_function this is the function to be used for the smoothing. The functions possible are:

1. `equal`
2. `vinokur`[14](Δs_{begin},Δs_{end})
3. `cubic`(Δs_{begin},Δs_{end})
4. `vin2cub`(Δs_{begin},Δs_{end},ratio)
5. `sin`
6. `-sin`
7. `cos`
8. `laura`(Δs_{begin},S_{max},fstr,ep0,fsh)
9. `func`(*filename*) or func(*array_variable*)

NOTE: To retain the existing distribution or existing cell sizes for the distribution functions that require control parameters, include the argument *keepcurrent* with the **smooth** command.

xyz[...] is the region of a grid block to be smoothed.

boundary_conditions are specified as Dirichlet or fixed grid boundary conditions. The possible values are specified by `dirichlet=(...)` and the control parameters are dependent on the direction of smoothing; if smoothing is to be done in the I-direction, Dirichlet BC's are possible at `jmin`, `jmax`, `kmin`, and `kmax`.

NOTE: If one of the Dirichlet BC's is missing, that edge will have a Neumann boundary condition in which the grid will be smoothed at that edge.

tfi *dimension domain {iterations=} xyz[...]*

where,

dimension is the dimension of the TFI. This can be either `2d` or `3d`.

domain is the physical or computational domain to be used for the TFI interpolants. The possible values for the argument can be `physical` or `parametric`.

iterations= this is the number of iterations to be performed to optimize the interpolants in three-dimensional TFI; 16 is sufficient.

> **NOTE:** This argument is only necessary when three-dimensional TFI is performed.

xyz[...] is the region of a grid block to be regenerated.

13.6 Programming Language

$\boxed{\textbf{quit}}$

This command is used to end all VGM scripts. Each script **must** have one. This command is also augmented by **stop**, **exit**, **halt**, **end**, **bye**, and **by**.

$\boxed{\# \ \ldots}$

This command is used to identify comment lines. It **must** be the beginning character on a line.

$\boxed{\ldots\backslash}$

This command is the line continuation marker. If a command and its arguments can not fit on a single 80 character line, the continuation marker allows the remaining arguments to be placed on the next line.

Chapter 14

Trouble Shooting and Errors

Since the VGM code is more of a language, there are numerous places that error checking occurs. Though not inclusive of all the possible errors that can be detected, the VGM code does identify as many as possible. This chapter lists all the possible errors that are detected and explains each.

The VGM code also writes a debug file, generated based on the UNIX process identification number (PID). The construct of the file name is:

```
VGM_debug-######
```

where, the #'s represent the PID.

The next sections are arranged in order of specifics:

Language Errors -	errors resulting from general language anomalies, including syntax and spelling.
Command Errors -	unique errors resulting from arguments of a specific command.
Redistribution Errors -	errors that are common to the **redist** and **smooth** commands.
Input and Output Errors -	errors that result from **read** and **write** commands.

There is no particular order to the errors listed in each section. Just the error and the meaning of it with respect to the command or commands.

14.1 Language Errors

```
***
*** ERROR: Number of consecutive lines skipped > 10.
***        Issuing QUIT command sequence.
***
```

The maximum number of blank lines allowed in VGM is 10. To use any more, causes this error to be displayed, and the code will gracefully stop. This was implemented because if the script does not have an finishing command such as **quit** the end of file is re-read and re-read.

```
***
*** ERROR: Domain not specified (parametric or physical)!
***
*** line:iline  cmdline(iline)
***
```

Those commands that require a domain argument must have that argument on the command line or this error will result. Usually the user has forgotten to add this argument.

```
***
*** ERROR: Type of interpolants not specified!
***
*** Command Line: cmdline(icmd)
***
```

The **copydist redist** and **smooth** commands require the bridging function to be used to generate the basis curves for all interpolations.

```
***
*** ERROR: Interpolation method not specified.
***
*** Command Line: cmdline(icmd)
***
          -or-
***
*** ERROR: Multiple occurrences of interpolation methods found.
***
*** Command Line: cmdline(icmd)
***
```

The interpolation basis type is required for all commands requiring the computation of arclengths to form basis functions for redistribution. The possible types are `linear` or `spline`.

```
***
*** ERROR: Incorrect limit specifications:
***
*** Specification: cmdpart(nuse)
***
```

The limits chosen for a grid block or an array variable have been exceeded. Check the limits of these storage types.

```
***
*** ERROR: Direction entered is not available:
***
*** Direction: cmdline(icmd)
***
```

Most commands require a direction of interpolation, computation or blending. The possible types are `i-direction`, `j-direction`, or `k-direction`. Any other direction chosen will result in this error message.

```
***
*** ERROR: Type of arc-length not specified.
***  IARC=iar
***
***
*** Command Line: cmdline(icmd)
***
          -or-
***
*** ERROR: Multiple occurrences of <arclength> found.
***  IARC=iar
***
***
*** Command Line: cmdline(icmd)
***
```

Each command that uses an arclength function can utilize a physical or computational domain dependency on the arclengths, except the **smooth** command. Those commands requiring the computation of the arclength need to have the type specified. The possible values are `arclength` for the physical domain, and `parametric` for the computational domain.

```
***
*** ERROR: Incorrect command on line icmd!
***
```

There are only 12 commands, including # for comment lines. Check the spelling of the command requested.

```
***
*** ERROR: File not found!
***
```

The file in a **read** or redistribution command does not exist. Check the spelling of the file name.

14.2 Manipulation Command Errors

14.2.1 ALLOCATE Command

```
***
*** ERROR: Core Variable chosen does not exist:
***
***  cmdpart(ivar)
***
```

This should be fairly straight forward. If you assumed an array variable exists and use it in specifying block limits, and the variable has never been defined via an **allocate** command, this error will identify which argument of the current command has that undefined variable. Check the spelling of the array variable and the spelling of the variable used in it's allocation.

14.2.2 BLEND Command

```
**
** POLE identified, switching to
**   J=
**   K=
**
```

This warning tells the user that a singularity has been detected on a face or in a volume grid that has requested the computing of the `physical` arclength parameter. This waring comes from the **blend** command, where the processes is trying to establish an arclength parameter space. To account for the singularity, the code first changes the index at which it computes the arclength, to search for a non-singular grid line. If it does not, the `physical` domain will be changed to `parametric` domain.

```
***
*** ERROR: Blending not performed.
***        Direction and limits are not conducive:
***        For direction, # to blend < direction MAX
```

The increment on the direction of the **blend** command produces intervals that are not equal. To properly blend multiple regions, the intervals resulting from the limits in a one dimensional direction blend must be equal. Check the increment.

```
***
*** ERROR: 1D Interpolation not possible with the following method:
***
```

When blending a variable in one dimension, the TFI and LARCS blending types are not allowed. This error can result from either requesting a non-one-dimensional interpolation type, or a syntax error in the blending type.

```
***
*** ERROR: Dimension of blending/interpolation not specified!
***
*** Command Line: cmdline(icmd)
***
```

The **blend** command requires the type of blending to be used, specified with the `interpolation=` argument.

```
***
*** ERROR: Extrapolation can not be done!
***
*** X(1)=x(1)   X(Idim)=x(idim)
*** U(J)=u(j)
***
```

This error results from using the `elliptic` interpolation type for the **blend** command. If the limits of the blend require extrapolation beyond the the limits of available data, elliptic extrapolation can not be done, only linear extrapolation is possible. Check the limits of the array variable and the arclength domain to be used if and only if the `physical` domain is in use.

```
***
*** ERROR: Not enough values to use for interpolation!
***
Number of values required: 2
Number of values given: n
```

When using the `linear` or `elliptic` interpolation types in the **blend** command, at least two data points have to be used, a beginning and ending point. This error usually results from choosing the increment of the blending zones to be too large. Check the limits of the array variable and the arclength domain to be used if and only if the `physical` domain is in use.

```
***
*** WARNING: No values need to be interpolated.
***
```

When using the `linear` or `elliptic` interpolation types in the **blend** command, at least one point should be interpolated. This error usually results from choosing the increment of the blending zones to be too small. Check the limits of the array variable and the arclength domain to be used if and only if the `physical` domain is in use.

```
***
*** ERROR: Input abscissas out of order.
***
```

When using the `linear` or `elliptic` interpolation types in the **blend** command, the dependent variable has to be in order, but the independent can vary wildly. This error usually results from crossed grid lines being chosen for a particular manipulation. Check the grid.

```
***
*** ERROR: Volume identified for 2D manipulation!
***
```

LARCS interpolation type is only available for `2d` and `3dp` blending. A volume zone has been requested, which can not be blended with LARCS. Check the interpolation type or the dimensionality of the **blend** command.

14.2.3 COMBINE Command

```
***
*** ERROR: Current combination of blocks will exceed the available
***        block limits!
***
***   Maximum Blocks: limblk
***   Blocks Needed: mxblkm + delmblk
***
```

The combine command works by increasing the block reference list by a specified number of blocks, given in the **combine** command, instead of making duplicate copies of a block. The new blocks placed in the reference list are done so by noting the counters to the positions in the physical coordinate arrays. This error results if the number of blocks to be duplicated causes the maximum number of blocks in the current VGM execution to exceed the maximum number of blocks available to the code.

14.2.4 COPYDIST Command

```
***
*** ERROR: Incorrect destination grid limit specifications:
***
*** Specification: cmdpart(nuse)
***
     -or-
***
*** ERROR: Number of sources does not match
***        number of destinations:
***
***     INDEX        Source       Destination
***     =====        ======       ===========
***
```

The **copydist** command requires the number of destination grid lines to be identical to the number of source grid lines. If this is not true, this error message will appear. Check the limits in the cross-direction to the direction of the copy (i.e. if the I-direction is the copying direction, the cross-directions are J and K).

```
**
** POLE identified in basis.
** Resulting points will have basis values.
**   J=j
**   K=k
**
     -or-
**
** POLE identified in basis.
** Resulting points will have basis values.
**   I=i
**   K=k
**
     -or-
**
** POLE identified in basis.
** Resulting points will have basis values.
**   I=i
**   J=j
**
```

The **copydist** command will identify pole boundaries in the direction chosen for the destination grid. If a pole boundary exists the resulting grid will not be changed from its original positions. This is just a warning to tell the user that a switch of operations has been done.

14.2.5 REDIST Command

```
***
*** ERROR: Incorrect New Block specifier:
***
***  cmdpart(iblk)
***
```

This error can be produced by the **redist** and **set** commands. For the **redist** command, the only two answers to the `newblock=` argument are "yes" and "no", and any thine else will generate this error message. The **set** command uses this error message to tell the user that you can not set a grid block equal to a single array variable. The set has to be to another grid block.

```
***
*** ERROR: Number of new points not specified!
```

```
 ***
 *** Command Line: cmdline(icmd)
 ***
```

In the **redist** command, the number of points to be placed along the basis curves must be specified, with the `points=` argument.

```
***
*** ERROR: No new block specifier conflicts
***        with other inputs.
***
*** Implied condition: ITOTAL=NEWPTS
*** ITOTAL=itot   NEWPTS=',newpts
***
    -or-
***
*** ERROR: No new block specifier conflicts
***        with other inputs.
***
*** Implied condition: JTOTAL=NEWPTS
*** JTOTAL=',jtot,'   NEWPTS=',newpts
***
    -or-
***
*** ERROR: No new block specifier conflicts
***        with other inputs.
***
*** Implied condition: KTOTAL=NEWPTS
*** KTOTAL=',ktot,'   NEWPTS=',newpts
***
```

The number of points specified in the direction of a redistribution does not match the number of points already in that direction for the limits chosen in the grid block specification. This is conflicting with the non-newblock specification; so the manipulation is not done. Check the limits of the grid block to be redistributed and the number of new points to be placed along that grid line; if these two limits are not identical this error results.

14.2.6 SET Command

```
**
** This feature is not available.
** It can be done by:
**
** allocate xvar[...]
** set xvar[...] = 0.0
** set xyz[1] = xvar y(xyz[1]) z(xyz[1])

         -or-

** allocate yvar[...]
** set yvar[...] = 0.0
** set xyz[1] = x(xyz[1]) yvar z(xyz[1])

         -or-

** allocate zvar[...]
** set zvar[...] = 0.0
** set xyz[1] = x(xyz[1]) y(xyz[1]) zvar
```

The feature being requested is to place a grid intrinsic on the left hand side of the equal sign in the **set** command. This is not allowed. To set a single coordinate of a grid block to a number, the code gives the user 3 different command sequences.

```
***
*** ERROR: Trying to assign a temporary variable
***        more than one value.
***
*** TEMPVAR = ctmpv(ntmpvar)(1:ncpertmpv(ntmpvar))
***
```

Temporary variables can only have one value. To try to set multiple values to a constant is impossible. Remember that temporary variables are constants in VGM.

```
***
*** ERROR: Core variable value type unrecognizable.
***
***  CMDPART=cmdpart(4)
***
```

The capability to assign a constant value to an array variable exists, but the constant has to be either a variable or a number. This error will result if the constant is not another array variable, constant, or a number. Check the spelling of the variable on the right hand side.

```
***
*** ERROR: General Math not allowed:
***
*** cmdline(1)
***
```

When using the **set** command, general equations of math are not allowed. The only types of values that can be set are constants, array variables, grid blocks and intrinsics. Each can have a "-" sign in front of the value, but that is all. This error can also be given if the variable or value on the right hand sign is not one of the possible values or if it is misspelled.

```
***
*** ERROR: No match of limits can be found:
***
*** cmdline(1)
***
```

The **set** command requires the indices on the left to match the indices on the right of the equal sign for block manipulations such as grids and array variables. If the limits do not match, one side does not have enough memory to be equated to the other.

14.2.7 SMOOTH Command

```
***
*** ERROR: Number of points to be used for 1D vector construction'
***        not specified.
***
*** Command line:
*** cmdline(icmd)
***
```

When smoothing a grid with the hermite vector interpolation, the order of the vector needs to be specified with `order=#`. If not this error will result from that argument not on the command line.

```
***
*** ERROR: Dirichlet boundaries not specified.
***
*** Command line:
*** cmdline(icmd)
***
```

When smoothing a grid with the hermite vector interpolation, the boundary condition argument (`dirichlet=(...)`) needs to be on the command line. If the user does not want any boundaries held fixed, the argument still has to appear, but no control words are to be placed inbetween the parentheses.

```
**
** WARNING: Distribution problems detected.
**          Locations written to fort.8 and fort.20
**
```

The `vinokur` and `cubic` distribution functions may not work with the specified control parameters. This can produce NaN or infinity for results. If this happens, try changing the limits of the smoothing or the sizes of the beginning and ending cells. Also check the length to which the cells are being applied. There may not be enough distance to accommodate the cell sizes or there may be too much.

14.2.8 TFI Command

```
***
*** ERROR: Number of 3DTFI iterations not specified!
***
*** Command Line: cmdline(icmd)
***
```

To do three-dimensional TFI, the number of optimization iterations needs to be specified with the `iterations=` argument. In two-dimensional TFI, there is no optimization.

14.3 Redistribution Errors

```
***
*** ERROR: Surface and redistribution control variable limits do not match:'
***
*** Surface:
***
***  Itot=itot
***  Jtot=jtot
***  Ktot=ktot
***
***
*** Control Variables: <vinokur & cubic>
***
***  Ltot(DSB)=ld_dsb   Mtot(DSB)=md_dsb
***  Ltot(DSE)=ld_dse   Mtot(DSE)=md_dse
***

***
*** Control Variables: <vin2cub>
***
***  Ltot(DSB)=ld_dsb   Mtot(DSB)=md_dsb
***  Ltot(DSE)=ld_dse   Mtot(DSE)=md_dse
***  Ltot(RATIO)=ld_ratio  Mtot(RATIO)=md_ratio
***

***
*** Control Variables: <laura>
***
***  Ltot(DSB)=ld_dsb   Mtot(DSB)=md_dsb
***  Ltot(DSE)=ld_dse   Mtot(DSE)=md_dse
***  Ltot(FSTR)=ld_fstr Mtot(FSTR)=md_fstr
***  Ltot(EP0)=ld_ep0   Mtot(EP0)=md_ep0
***  Ltot(FSH)=ld_fsh   Mtot(FSH)=md_fsh
***

***
*** Control Variables: <vin2cub>
***
***  Ltot(NEWPOINTS)=ld_junk  Mtot(NEWPOINTS)=md_junk
***

***
***       <<< AND/OR>>>
***
```

```
***  JTOT != LTOT   -or-  KTOT != MTOT
***  ITOT != LTOT   -or-  KTOT != MTOT
***  ITOT != LTOT   -or-  JTOT != MTOT
***
```

When specifying the control parameters for a distribution function argument in the **redist** and **smooth** commands, if arrays are used, each array used as a control parameter has to have the same dimensional limit in the cross-directional indices. Note that each distribution function has different control parameters that are checked; hence the extent of the above error message. The code will only print out the message that identifies the function chosen.

```
***
*** ERROR: Cross-direction limits of core variable do not match grid limits:
***
*** Jtot=jtot   Mtot=mtot
*** Ktot=ktot   Ntot=ntot
***
          -or-
***
*** Itot=itot   Ltot=ltot
*** Ktot=ktot   Ntot=ntot
***
          -or-
***
*** Itot=itot   Ltot=ltot
*** Jtot=jtot   Mtot=mtot
***
```

When specifying the control parameters for the distribution function arguments of the **redist** and **smooth** commands, the index limits of the face or point identified by the cross-directional indices to the direction of the redistribution must match the index limits of the array or constant variables. If there is no match of these limits, the code can not perform the redistribution because there is not enough data for the chosen function. This error usually results from indices being incorrect in the grid block specifier, or misspelled variable name or even an undefined variable. The latter is more difficult to track because an **allocate** may have created a misspelled array variable. Check the indices of all variables, and grid blocks being referenced and check the dimensions.

```
***
*** ERROR: Distribution function not found.
***
```

The **redist** and **smooth** commands require a distribution function. If one is not selected, this error message will appear.

```
***
*** ERROR: Core Variable chosen and Block limits do not match:
***
***  Block Limits:
***    IDIM=idima(nblku)
***    JDIM=jdima(nblku)
***    KDIM=kdima(nblku)

          -or-

***    Itot=itot
***    Jtot=jtot
***    Ktot=ktot
***
      -and-

***  Core Variable Limits:
***    LDIM=ldim(mcorv)
***    MDIM=mdim(mcorv)
***    NDIM=ndim(mcorv)

          -or-

***    Ltot=ltot
***    Mtot=mtot
***    Ntot=ntot
***
```

When specifying the index limits for the **blend** command with interpolation in the `physical` domain, the limits of the grid block **must** match the limits of the blending, including the increments. Otherwise, the amount of data needed to compute the arclengths is incorrect.

```
***
*** ERROR: File for variable not read.
***
***        GRIDGEN format assumed.
***
```

If a variable that represents a file name, used as a control parameter for the distribution function of a **redist** or **smooth** commands, does not exist or is in the incorrect format, this error will result. Each of the distribution commands can hold at most, a 2D surface of control parameters for a distribution function. The format of these control parameters is GRIDGEN because it is surface based. Check the format of the file being used to load a control variable.

```
***
```

```
*** ERROR: Specified limits of core variable must be in 2D computational space!
***        One of the following has to be 1:
***
***  Ltot=ltot
***  Mtot=mtot
***  Ntot=ntot
***
```

If a variable used as a control parameter for the distribution function of a **redist** or **smooth** command does not represent at most a 2D surface, this error will result. Check the limits of the variable being used to specify the control parameters.

```
***
*** ERROR: Problem w/internal-read.
***
*** CVAR=cvar(nvar)
*** FILEU=fileu
*** NC=ncf
***
```

If a constant is used for a control variable to a distribution function in the **redist** or **smooth** commands, and the constant is not discernible due to characters other than "E" for exponential notation, an error will result. This error will also result if the constant is not determined to be an array variable or file name. Check the constant requested.

```
***
*** ERROR: Unrecognizable distribution.
***
*** DIST option: ',dist
***
```

There are only 9 different distribution functions available. The one you have requested does not exist. This usually occurs if the distribution function is misspelled.

```
***
*** ERROR: Could not correct poor parameterization:
***
*** I-Direction REDIST
*** MP1,MP2=mp1,mp2
*** JPT,DSnew(JPT)=jpt,dsnew(jpt)
*** i,j,k: i,j,k
***
     -or-
***
*** ERROR: Could not correct poor parameterization:
***
```

```
*** J-Direction REDIST
*** MP1,MP2=mp1,mp2
*** JPT,DSnew(JPT)=jpt,dsnew(jpt)
*** i,j,k: i,j,k
***
     -or-
***
*** ERROR: Could not correct poor parameterization:
***
*** K-Direction REDIST
*** MP1,MP2=mp1,mp2
*** JPT,DSnew(JPT)=jpt,dsnew(jpt)
*** i,j,k: i,j,k
***
```

When the `spline` function is used to construct the basis function for a redistribution, since the spline is unclamped, the resulting curve could produce negative volumes by reversing the direction of the basis curve. If this occurs, the VGM code will attempt to correct it by isolating the region that is bad and redistributing it to correct the curve. The correction changes the parameterization slightly, but does reduce the risk of generating negative cells or volumes. This error will tell the user if the code can not re-parameterize the basis function to alleviate the possible generation of negative cells or volumes. Try using `linear` basis function generation.

14.4 Input and Output Errors

```
***
*** ERROR: TECPLOT binary file can not be read!
***
```

The only format available to read TECPLOT data files is ASCII. Rewrite the TECPLOT file as ASCII to get VGM to read it.

```
***
*** Nothing read due to flags not set.
***
```

Although the input command has defaults, some of the argument types may be specified incorrectly or not at all. Check to make sure there is a file to be read or written.

```
***
*** ERROR: Input command not found.
***
*** INPUT COMMAND: cmdpart(npart)(1:ncperpart(npart))
***
```

The **read** command has found an argument that makes no sense. Usually VGM will try to interpret this as a file name. If it is not, check the spelling of the argument.

```
***
*** ERROR: Data block form incorrect or file not complete!
***
*** BLOCK: nblk
***
```

During the input phase of the **read** command, an end-of-file (EOF) was found or the data types are incorrect (i.e. trying to read a floating point number into an integer variable). Check the format of the file.

```
***
*** ERROR: File to be appended not found.
***
*** FILE: fileu(1:nc)
***
*** Above filename being set up anew.
***
```

The file requested to be appended does not exist. The VGM code will create this file for the output of data. If this action is not adequate, check the spelling of the file name to be appended.

```
***
*** ERROR: TECPLOT variable limit mismatch:
***
***  I2=i2     LDIM=ldim(mcorv)
***  J2=j2     MDIM=mdim(mcorv)
***  K2=k2     NDIM=ndim(mcorv)
***
```

When data is being written in TECPLOT form, the array variables being written must have at least the number and range of indices available for the write. If the limits of writing a variable exceeds the limits of that variable, this error will result and no data will be written.

```
***
*** ERROR: Grid System not specified for Solution output.
***
```

The solution argument of the input and output commands requires a grid system number to attach and reference the solution data to a grid. Change the solution argument to reflect the grid system number.

```
***
*** ERROR: Mach, Alpha, Re, Time does not exist for solution data set: ngsys
***
```

The flow constants in the PLOT3D styled solution file are not in the file or are not readable. Check the solution file to verify the correct data is in the header.

```
***
*** ERROR: Number of variables for solution data is incorrect.
***        Only 5 variables can be written.
***
```

The PLOT3D style of solution data file can only support 5 flow variables. To request more, another solution file needs to be written.

```
***
*** ERROR: Grid System and Block does not have Solution variables.
***
```

To reference flow variables in the writing of data into the PLOT3D style, they must be of the form discussed in section 4.1 or listed separately in the solution argument.

```
***
*** ERROR: Grid System and Block does not have LAURA variables.
***
```

The grid system number chosen to write out a LAURA restart file does not have flow variables attached to it. Check the grid system number in the LAURA argument.

```
***
*** ERROR: Conflicting arguments:
***
*** TECPLOT and Switching of coordinates is not possible.
***
```

The `switch` argument can not be used on array variables, so it can not be used with the TECPLOT output style.

```
***
*** ERROR: Conflicting arguments:
***
*** Core variable output and Switching of coordinates is not possible.
***
```

The `switch` argument can not be used on array variables, just physical grid blocks.

```
***
*** ERROR: Conflicting arguments:
***
*** Binary TECPLOT files are not possible.
***
```

Currently, only the ASCII mode of TECPLOT data style is available for reading and writing.

```
***
*** ERROR: Conflicting arguments:
***
*** Different number of TECPLOT variables for an append.
***
```

When appending a data set to a previously opened file, the number of variables must match the number of variables in the file. The VGM code does not keep track of multiple files, only the previously written TECPLOT file. If the number of variables requested to be written does not match the number already written, this error message will result. Check the number of variables for each file to be appended and be sure that, that number is identical.

```
***
*** ERROR: Conflicting arguments:
***
*** Solution output of no solution data.
***
```

To output solution data, solution data must exist. This could be caused by not referencing the correct grid system number. Check the grid system number and the solution argument.

```
***
*** ERROR: Incorrect source grid limit specifications:
***
*** Specification:  cmdpart(npart)(1:ncperpart(npart))
***
*** I=i1 to i2
*** J=j1 to j2
*** K=k1 to k2
***
```

The limits chosen for a grid block or an array variable have been exceeded. Check the limits of these storage types.

```
***
*** ERROR: Output command not found.
***
*** OUTPUT COMMAND: cmdpart(npart)(1:ncperpart(npart))
***
```

The **write** command has found an argument that makes no sense. Usually VGM will try to interpret this as a file name. If it is not, check the spelling of the argument.

```
***
*** ERROR: Grid System number to output, not valid.
***
*** GRID SYSTEM REQUESTED: ngsys
***
```

The grid system being identified for output in a grid block or solution argument or an array variable is incorrect. Check the grid system number requested.

References

[1]S. J. Alter and K. J. Weilmuenster, "The Three-Dimensional Multi-block Advanced Grid Generation System (3DMAGGS)," NASA TM-108985, April 1993.

[2]J. P. Steinbrenner, J. R. Chawner, and C. L. Fouts, "The GRIDGEN 3D Multiple Block Grid Generation System," Wright Research and Development Center Report WRDC-TR-90-3022, October 1989.

[3]R. W. Noack and D. A. Anderson, "Solution Adaptive Grid Generation Using Parabolic Partial Differential Equations," AIAA paper 88-0315, January 1988.

[4]W. M. Chan, I. Chiu, and P. G. Buning, "User's Manual for the HYPGEN Hyperbolic Grid Generator and the HGUI Graphical User Interface," NASA TM-108791, October 1993.

[5]P. A. Gnoffo, "An Upwind-Biased Point-Implicit Relaxation Algorithm for Viscous, Compressible Perfect-Gas Flows," NASA Technical Paper 2953, February 1990.

[6]I. Amtec Engineering, "Tecplot: version 5 User's Manual," Amtec Engineering publication V5.0/92-14, January 1992.

[7]P. P. Walatka, P. G. Buning, L. Pierce, and P. A. Elson, "PLOT3D User's Manual," NASA TM 101067, March 1990.

[8]R. L. Sorenson and S. J. Alter, "3DGRAPE/AL: The Ames/Langley Technology Upgrade," NASA CP-3291, pp. 447-462, May 1995.

[9]R. W. Walters, D. C. Slack, W. M. Eppard, M. Applebaum, C. B. Fury, A. G. Godfrey, and W. D. McGrory, *GASP 3: User Manual.* Blacksburg, VA: Aerosoft, first ed., May 1996.

[10]V. N. Vatsa and B. W. Wedan, "Development of a Multigrid Code for 3-D Navier-Stokes Equations and Its Application to a Grid-Refinement Study," vol. 18, pp. 391-403, Computers & Fluids, June 1990.

[11]P. P. Walatka, J. Clucas, R. K. McCabe, T. Plessel, and R. Potter, "FAST User Guide," NASA Ames Research Center RND-93-010, June 1993.

[12]P. Raj, J. E. Brennan, J. M. Keen, K. K. Mani, C. R. Olling, J. S. Sikora, and S. W. Singer, "Three-Dimensional Euler Aerodynamic Method (TEAM)," Flight Dynamics Laboratory, Wright Research and Development Center AFWAL-TR-87-3074, June 1987.

[13]J. Samareh, "GridTool: A Surface Modeling and Grid Generation Tool," in *Surface Modeling, Grid Generation, and Related Issues in Computational Fluid Dynamic (CFD) Solutions*, pp. 821-831, NASA CP-3291, first ed., May 1995.

[14]M. Vinokur, "On One-Dimensional Stretching Functions for Fininte-Difference Calculations," NASA CR-3313, 1993.

[15]K. J. Weilmuenster, P. A. Gnoffo, F. A. Greene, C. J. Riley, H. H. Hamilton III, and S. J. Alter, "Hypersonic Aerodynamic Characteristics of a Proposed Single-Stage-to-Orbit Vehicle," AIAA Paper 95-1850, June 1995.

[16]S. J. Alter and F. M. Cheatwood, "Elliptic Volume Grid Generation for Viscous Computations in Parametric Design Studies," AIAA Paper 96-1999, June 1996.

[17]D. D. R. Olynick, "Importance of 3-D Grid Resolution and Structure for Calculating Reentry Heating Environments," AIAA Paper 96-1857, June 1996.

[18]S. J. Alter and F. M. Cheatwood, "Volume Grid Expansion Techniques for Computational Fluid Dynamic Algorithms," AIAA Paper 96-0029, January 1996.

[19]A. S. J. and K. J. Weilmuenster, "Cell Volume Control at a Surface for Three-

Dimensional Grid Generation Packages," in *Software Systems for Surface Modeling and Grid Generation* (R. E. Smith, ed.), pp. 273–298, NASA CP–3143, 1992.

[20]B. K. Soni, "Two– and Three–Dimensional Grid Generation for Internal Flow Applications of Computational Fluid Dynamics," AIAA Paper 85–1526, 1985.

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE April 1997	3. REPORT TYPE AND DATES COVERED Contractor Report

4. TITLE AND SUBTITLE The Volume Grid Manipulator (VGM): A Grid Reusability Tool	5. FUNDING NUMBERS Contract NAS1-96014 Task Order DG02 WU 242-80-01-01
6. AUTHOR(S) Stephen J. Alter	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Lockheed Martin Engineering & Sciences Company Langley Program Office Langley Research Center, Mail Stop 371 Hampton, VA 23681-0001	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Langley Research Center Hampton, VA 23681	10. SPONSORING/MONITORING AGENCY REPORT NUMBER NASA CR-4772

11. SUPPLEMENTARY NOTES

Langley Technical Monitor: K. J. Weilmuenster

12a. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified - Unlimited Subject Category 34	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This document is a manual describing how to use the Volume Grid Manipulation (VGM) software. The code is specifically designed to alter or manipulate existing surface and volume structured grids to improve grid quality through the reduction of grid line skewness, removal of negative volumes, and adaption of surface and volume grids to flow field gradients. The software uses a command language to perform all manipulations thereby offering the capability of executing multiple manipulations on a single grid during an execution of the code. The command language can be input to the VGM code by a UNIX style redirected file, or interactively while the code is executing. The manual consists of 14 sections. The first is an introduction to grid manipulation; where it is most applicable and where the strengths of such software can be utilized. The next two sections describe the memory management and the manipulation command language. The following 8 sections describe simple and complex manipulations that can be used in conjunction with one another to smooth, adapt, and reuse existing grids for various computations. These are accompanied by a tutorial section that describes how to use the commands and manipulations to solve actual grid generation problems. The last two sections are a command reference guide and trouble shooting sections to aid in the use of the code as well as describe problems associated with generated scripts for manipulation control.

14. SUBJECT TERMS: Computational Grids; Grid Generation (mathematics); Viscous Grids; Computational Fluid Dynamics; Numerical Techniques	15. NUMBER OF PAGES 148
	16. PRICE CODE A07

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102