Basis3D - a Platform for Development of Multiblock PDE Solvers
β - release

Michelsen, Jess A.

Publication date:
1992

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
Basis3D - a Platform for Development of Multiblock PDE Solvers

β - release, December 1992

Jess A Michelsen
Abstract

The Basis3D code is intended as an aid to the developer of three-dimensional PDE solvers for continuum problems, such as transport problems, electrical or electromagnetic field problems, and mesh generation. Supporting finite element, finite volume, and finite difference usage, Basis3D can be employed with vertex centered and cell centered discretizations. Face centered variable arrangements, as employed with staggered grids, are not supported in the present version. Face-centered storage of data, that need no communication is, however, possible. Also not supported is the use of co- and contravariant tensor variables of order higher than zero, as they may change physical significance across block-interfaces. Space discretization is of block-structured type, where each block consists of \( n^3 \) cells, \( n \) conveniently being a multi-grid number. The blocks must be conforming, that is the coordinate lines are continuous across block-interfaces. A set of associated library routines support the use of Domain Decomposition and Multigrid techniques. All communication between multigrid levels and between mesh blocks are taken care off by these routines. The array structure allows the use of direct addressing in most parts of the potential user applications. The Basis3D program is run with an input file defining the grid blocks as a pre-processor before mesh-generation, problem solutions, etc. The output is a file containing a large number of tables and masks to aid the inter-block communication. Data-structures of user-defined boundary conditions and variables are proposed.
Contents

1 Introduction 5

2 Topology concept 7

3 Implementation model 11
   3.1 Arrays ............................................. 15
   3.2 Multigrid handling ............................... 17

4 Running applications 22
   4.1 Basis3D input ...................................... 22
   4.2 Basis3D output .................................... 24
   4.3 Prototype application ............................ 25
   4.4 Library routines .................................. 26
      4.4.1 I/O routines .................................. 26
      4.4.2 Interblock communication routines ........... 27
      4.4.3 Multigrid tools ............................... 29
      4.4.4 Miscellaneous tools ......................... 32

5 Ongoing and contemplated extensions 34

A The Basis3D include files ......................... A-I
   A.1 Params.inc ....................................... A-I
   A.2 Attrib.inc ....................................... A-IV
   A.3 Blockcon.inc .................................... A-IV
   A.4 Blockdef.inc .................................... A-V
   A.5 Cvtables.inc ..................................... A-V
   A.6 Cvmasks.inc ..................................... A-VIII
   A.7 Decode.inc ....................................... A-VIII
   A.8 Libspace.inc ..................................... A-IX
   A.9 Vertices.inc ..................................... A-IX

B Setting up logical masks ......................... B-I
   B.1 Finite differences with Dirichlet BC ............. B-I
   B.2 Cells touching an exterior face .................. B-III
Chapter 1

Introduction

Over the years, a considerable number of different PDE solvers have been developed at the Department for Fluid Mechanics. For each of these codes, one or more persons have undertaken the tedious task of defining (yet another) data-structure and debugging their way through all sorts of problems.

Apart from the fact that the amount of work spend on coding and testing the book-keeping is quite large, this modus operandi results in a low degree of coherence between different code-development projects. The integration of different codes thus often becomes a difficult task, and might for that reason often be abandoned.

During the summer 1991 a decision in principle was made, to define a data-structure that could be employed by all code-developers at the Department. The Basis3D is an attempt to realize this decision.

When defining a data-structure and mesh-type, the trade-off between flexibility (geometric and in other senses) versus transparency, simplicity, and code efficiency must be taken into account. Here, the authors opinion is that simplicity and transparency, within reason, come before maximum flexibility. Taking a view over the projects already carried out at the Department, and those anticipated in foreseeable future, one finds no demand for very high geometric flexibility, such as provided by unstructured triangular/tetrahedric meshes. In fact, the simplest possible multi-block approach, namely using geometrically conforming blocks, each consisting of $(2^n)^3$ cells, readily yields the necessary degrees of freedom. The block sidelength $(2^n)$ may be increased to an optimal size in order to decrease the amount of communication at the dispense of flexibility, and vise versa. Sidelengths other than powers of two can of course be employed. However, the coarsest mesh in an eventual multigrid strategy will be finer, thus potentially reducing the convergence speed.

The choice of this simple approach has obvious advantages. First of all, direct addressing can be employed throughout the crunching part of the code. Load balancing on parallel architectures is greatly facilitated. No interpolation of fluxes etc. at block boundaries is needed, as is the case for non-conforming blocks. Perhaps most important is that a high degree of transparency is preserved.

The use of fixed-size cubic blocks instead of allowing slab-shaped blocks of $ni \cdot nj \cdot nk$ cells may at first glance seem a disadvantage. However, if the developer wishes to employ a foreign mesh-generator that generates slab-blocks, the slabs will just have to be cut into cubes, prior to use with the Basis3D structure. Moreover, interface inconsistencies related to mismatching cell numbers are avoided. This is by the way one of the known trouble-spots with many existing 3D multi-block codes.
Again taking a simple approach, only inter-block communication, where the source and the target arrays are the same, is supported. This excludes the use of mesh-aligned variable components, e.g. co- and contravariant velocity components. For this reason, face centered variable arrangements, such as those employed in staggered mesh arrangements are not supported. If desired, such multi-array communications may be included in the libraries. Both communication of cell-vertex (CV) and cell-centered (CC) data is supported by the present implementation.

The Basis3D code is typically employed in a 3-step manner. Prior to running Basis3D, a mesh-file containing the vertex coordinates is generated. Basis3D is then run as a pre-processor. It reads the mesh-file, generates a file containing tables for communication, and validates the mesh-file. Hereafter, user application(s) are run, supported by the Lib3D library containing communication routines and other general purpose tools. The common structure of different applications should make desired code-integration fairly straight-forward.

It has been decided to employ Basis3D as a pre-processor, and hereby accept the reading/writing of large files, in favour of integrating the code with the user applications. The most important reason for this is, that the vast majority of the Basis3D code is concerned with non-vector logical and integer operations. Hence, the Basis3D should preferrably be run on a sequential machine, while the choice of architecture for the applications is free. Only the communication routines, which are rather simple, needs to be tuned for a new architecture.
Chapter 2

Topology concept

The topology recognition of Basis3D follows a rule-based concept, as opposed to the hashing-table based algorithms employed with triangular mesh generators. The concept of a topology file, as introduced by Weatherill et al. [1, 2], holding for each block face, the neighbour face number, relative face orientation, and block number, is adopted. The present concept is, however, extended to include point-wise communication tables in the topology file. This way, it is possible to keep the associated communication routines extremely simple, in most cases only containing a set of nested loops. The action is then locally controlled by a set of index pointers and a logical mask. On data-parallel machines, in Fortran 90, the block-block transfer may consist of a single `where` - `endwhere` construct.

In contrast to a number of investigators, e.g. Shaw et al. [3], who stores only the data that belong to a block, the present concept employs an extra layer of data points outside the block. Also the NUGGET mesh generator by Coleman and Brabanski [4] employs storage of in-block data only. In order to facilitate the indirect addressing, the block is treated in 27 parts, namely the interior, 6 faces, 12 edges, and 8 corners. The authors advocate, that in 3D the memory and data-transfer overhead using an 'augmenting' layer outside the block is too high. Instead, the separate treatment of 27 objects is employed in order to vectorize the code.

These arguments are valid for vector architectures and certain parallel architectures. However, for most massively parallel systems, the SIMD types in particular, separate treatment of different parts of otherwise contiguous blocks will be heavily penalized by processor idling.

The present approach will perhaps run somewhat less efficient on vector processors. On the other hand, it is possible to write the number-crunching code in a data-parallel sense, which is the easiest to both vectorize and parallelize. Moreover, on the Connection Machine (CM), the latter approach saves router communication (the most general, and slowest) proportional to the number of operations involving field data! In addition, the CM automatically increases array sizes to fit the number of processor elements, making the above memory overhead arguments invalid.

The most evident advantage of the present approach, as opposed to those adopted by Shaw et al. and by Coleman and Brabanski is, that the task of data transfer addressing can be taken care of by pre-defined routines. Thus, instead of using indirect addressing throughout the code, the application programmer needs only to know when to communicate data, not how.

A few simple rules serve as the base for generating block topologies. First of all,
mesh points that are intended to coincide, must have exactly the same coordinates, as it is the coordinates, that are tested in order to interpret the topology. A block face may have only one opposed block face, which may be itself, see fig. 2.1. If two opposed block faces do not coincide over their entire surface, the coinciding points must still be aligned, see fig. 2.2. Two block faces are only accepted as coincident, if at least one of the four points near the corners, shown in fig. 2.3 coincide.

Figure 2.1: A blockface may at most have one opposed blockface, which may be itself.

Block faces, and thus block edges, are allowed to be collapsed or partially collapsed, within some restrictions. The most important restriction is, that a face (or part of face) is only allowed to collapse in one direction at a location, see fig. 2.4. However, it may collapse in two different zones in different directions. Two different parts of a block face may only collapse into the same line, if both parts include an edge over the entire collapsed stretch. Parts of two (or more) block faces may also only collapse into the same line, if both (or all) part-faces include an edge over their entire stretch.

The consequence of disobeying these restrictions would be that coincident points would be recorded as separate, thus the solution at one physical point would become non-unique. In order to safe-guard the user, Basis3D will check if any of the recorded point-clusters coincide, and report.

By the way, in this paper, the phrase cluster has a different meaning than usual. Whereas a cluster normally is a group of similar objects being closely spaced, the phrase here refers to a group of objects, e.g. mesh points, that literally coincide.
Figure 2.2: Mesh lines in opposed blocks must be aligned, even if faces are only partially connected.

Figure 2.3: If at least one of the depicted four points on a face coincides with the corresponding point on the opposed face, faces are connected.
Figure 2.4: Examples of legal and illegal collapses. A face is only allowed to collapse in one direction at a location.
Chapter 3
Implementation model

The Basis3D code presently consists of 77 code-segments (subroutines, functions, block data and main program), whereof 34 are generalized and included in the library Lib3D for user support. Here, the top levels of the Basis3D code structure will be presented. The processing and input/output of data is subdivided in 6 steps, each associated with a subroutine call, as sketched in fig. 3.1.

Figure 3.1: Top part of the Basis3D program structure.

Subroutine ReadMesh reads the mesh-file, while WriteMesh writes the same file upon completion. The sub-structure WriteCom constructs the communication file. These three sub-structures all belong to the Lib3D library.

Subroutine Geometry with sub-structure, see fig. 3.2, first computes the appropriate coordinate resolution to be employed in the coincidence-testing. The coordinates on edges, on faces, or in the interior of the blocks may be given as zero in the user-specified mesh-file. It is tested if any of the block-segments are zero. If this is the case, the coordinates on the actual edges, faces, or interiors are computed by
transfinite interpolation. The internal orientation of each block is then tested (the coordinates must define a right-hand system). The number of admissible multi-grid levels is computed, based on the array-sizes given and the block-size of the user-specified mesh. Finally, the coordinates and attributes are injected to the coarser multi-grid levels.

Subroutine \texttt{ConnectBlocks} with sub-structure, see fig. 3.3, determines the block-topology. The connected objects are registered, faces first, then edges, and finally corners. For each blockface, 6 for each block, the opposed block number, the opposed blockface number, and its surface orientation relative to the present blockface is tabellcd. Also, the internal orientation of the opposite block is tabellcd. These tables are part of the data written to the communication file. Moreover, tables for coinciding edges and corners are generated. These are employed only within the Basis3D code.

Subroutine \texttt{ComTables} with sub-structure as shown in fig. 3.4 computes all the point-based tables. A complete set of tables is prepared for each multi-grid level. This sub-structure is naturally the most involved part of the Basis3D code.

In CVinit the cell-vertex tables for one multigrid level are initialized. The coinciding-corner table, generated in \texttt{CornConn} is then scanned (CVcorn). For each corner-cluster, the adjacent block edges are tested to find if any part of the edges coincide with the corner-cluster. The entire vertex cluster is then labelled in the cluster array-structure. For MG level 1, the arrays of this structure are cli1, clj1, clk1, clblci holding the indices \((i,j,k,iblock)\) of each point in the cluster. The array cltyp1 holds the number of points in the cluster. Using these arrays, it is possible to collect information for the entire cluster. In order to find which cluster a given point belongs to, an additional array clpnt1 is constructed. Following the naming conventions, which are explained later, the corresponding array names for the coarser MG levels are determined by changing the appended digit.
After scanning the coinciding-corner table, the similar coinciding-edge table is then scanned (CV\text{edge}). Here, each edge-point is treated in succession. First, the rest of the edge and the eventually coinciding edges are tested for point-coincidence. If no coincident points are found along the edges, the adjacent block faces are scanned. It is in order to make this logic tractable, that block faces are not allowed to collapse in two directions at one point. The point clusters registered during the edge scan are entered in the same array structure as those found during the preceeding corner scan.

Finally, the block faces are scanned (CV\text{face}). If clusters are found, they are registered in the above mentioned arrays. If a given face-point only coincides with one point on an opposed face it is, however, not entered in the cluster table, as this would not be very efficient. Instead, entries for all points that coincide with one or more other points are made in a different array-structure, the image-structure. The names for MG level 1 are imacv1, jmacv1, kmacv1, bmacv1 for indices. Since the arrays are employed with direct addressing, two additional arrays copcv1 and sumcv1 are constructed. They are logical masks that tell if the copy and sum functions, respectively, should be carried out for the actual point. Copying of data within the above registered clusters are done using this structure, while the summation of data from a cluster needs both the special cluster-arrays and the image-arrays.

During computations, data needs to be transferred to the extra layer of points, outside the block face. The image structure is employed for this purpose. Hence, for the points outside the block, the indices of corresponding mesh points inside the neighbouring block are entered, see fig. 3.5.

The index pointers for the edges of the extra layer (across the block edges) are calculated in CV\text{diag}. For irregular clusters, see fig. ref(fig:irreg), special techniques are applied. The molecules thus obtained for the different points in a cluster are in general different. Hence, the first point of a cluster and the corresponding cluster is employed, while data for the remaining cluster is obtained by the copy functions. This is reflected in one of the mask examples of appendix B. An interior cluster of 4 points is, by the way, always a regular cluster.

In CV\text{corr}, certain corrections of the image structure are made for the points
CHAPTER 3. IMPLEMENTATION MODEL

![ComTables Diagram]

Figure 3.4: Top part of the ComTables sub-structure. This part of the code prepares a complete set of communication tables for each multigrid level. The sequence of substructures shown is called once for each MG level.

belonging to clusters. Mesh singularities are detected in $CV_{sing}$. The radial and axial directions of the singularity, in the transformed space, are determined. The value of the attribute of each point in the cluster is changed to reflect the radial/axial directions. This information is important for finite difference use, but is mostly not needed for finite volume use.

The vertex attributes given by the user in the mesh file are checked in $CV_{chk}$. It is tested if point designated as boundary points are really on a boundary, and so forth.

While the above sketched construction of the cell vertex topology is very complex, the cell centered topology is much easier to establish. The tables for communication of cell centered data are constructed in $CC_{face}$. The array names for level 1 are $imacc1$, $jmac1$, $kmacc1$, $bmacc1$, $copcc1$. Only the indices of the cell volumes across the cell faces are needed, as cell centers can not coincide.

As seen in fig. ref(fig:ccirreg), the image cell diagonally across the block edge may be non-unique. For the calculation of cell face fluxes, a different cell is needed for the $\xi$ and $\eta$ fluxes in the figure. This is the reason for using the shape selected for the Surface classes (to be discussed lateron). For cell centered calculations, where cross derivative terms appear, the flux calculation is segregated according to face direction. Before the calculation in each direction, the appropriate image-cell data should be copied into the cross-edge locations. This is done by a library routine.

It now only remains to write the communication tables on file, and write the (perhaps interpolated and certainly validated) mesh data back to the mesh file.
3.1 Arrays

The arrays employed in the Basis3D and the applications based upon it, may be classified in different ways. Two characteristics are here mentioned, namely the array shape and the array class. The array shape is (in Fortran) determined a priori, while the array class is defined by the type of data being kept in the array.

The number of different array shapes is here kept to the absolute minimum. Two different array shapes are primarily being used. The array shape to be employed for field-data is

array(ni,ni,ni,nb) addressed as array(i,j,k,iblock)

where (i,j,k) are the indices in the blocks $\xi, \eta,$ and $\zeta$ directions. In the fourth dimension, iblock refers to the block number.

This shape might be used for data only defined at block boundaries. However, another shape will be employed,

array(ni,ni,6,nb)

see fig. 3.8. This shape is addressed according to

array(j,k,1,iblock), array(j,k,2,iblock),
array(k,i,3,iblock), array(k,i,4,iblock),
array(i,j,5,iblock), array(i,j,6,iblock),

where (i,j,k) still refer to the above block indices.

Other array shapes may be defined by the user. For the sake of integrability, however, it is advisable to employ array shapes that keep one, two, or all three of the in-block indices in the first dimensions, along with the user defined dimensions, like
As mentioned above, the array classes are defined by the data being kept in the associated arrays. Hence, the types of operations valid for a specific array are determined by the arrays class. A number of array classes are defined in the present context, see table 3.1. More classes may be added by the application programmer. As two array shapes are primarily used, it is obvious that different array classes share array shape. Hence, it is possible for an array, preferably a workspace, to change class within a program.

The Vertex and Center classes are used for cell-vertex and cell-centered data, respectively. Presently, the communication routines are valid only for these two classes. The numbering in of CC data, relative to CV data is shown in fig. 3.9. It is emphasized, that the communication routines treat data as scalar fields. Hence, data with an implicit grid orientation will in general not be communicated correctly. The control functions used with the Poisson mesh generator is an example of such data. Special communication treatment of grid oriented data may at a later stage be contemplated.

The XiFace, EtaFace, and ZetaFace classes are employed for grid-directed data located at cell faces. If the data are defined on the cell faces with normal in the \( \xi \)-direction, the associated array is of the XiFace class. In the same sense, grid-directed data defined at cell edges are kept in arrays of the XiEdge, EtaEdge, or ZetaEdge classes. Presently, communication of these classes is not supported, for the reason that the physical significance of the grid-directions may change across the block-boundaries.

The three Surface classes share array shape. Similar to the above mentioned classes, arrays belonging to the CVSurface class is employed for cell vertex data defined on the block boundaries only, data for cell centers just outside or just inside the blockfaces are kept in arrays of the CCSurface class. Cell face centered data, i.e.
boundary conditions for CC based discretizations, are kept in CFSurface arrays. The communication tables for CC communication are of the CFSurface array class.

For all of the above classes, the data indexing follows the rules outlined in fig. 3.9. Moreover, for the classes of shape \((ni, ni, 6, nb)\) the above rule for numbering and orientation of faces applies.

The Cluster class is employed for the cluster tables employed for the summation of cell-vertex data at point clusters. The arrays of this class will only be of interest for application programmers when setting up logical masks. This subject will be addressed later.

The Projector class is employed, when a set of data is applied for all layers of all blocks. This way, a fixup of boundary points may in some cases be simplified.

All of the above array classes may of course be split in subclasses, where each subclass is concerned with a specific data type. Three data types are presently employed with the array classes mentioned here. They are integer*2, logical*1 and real*8.

### 3.2 Multigrid handling

The above described array shapes are different from the shapes normally employed for multigrid purposes. In most codes, data associated with the different levels in the multigrid hierarchy are stored in the same array, using a different offset for each multigrid level. However, if data-parallel architectures are considered, this is not an effective strategy. For instance, on the Connection Machine 2, the load balancing is based on the arrays. The array is simply distributed over the processor elements. Hence, for the above proposed single array storage, part of the processors are employed for the finest level, while only a very small portion of the processors
Figure 3.8: Boundary data are stored in 6 layers per block. Index variations are shown in the figure.

Table 3.1: The present array classes of Basis3D. The in-block index range in each grid direction is denoted by either $V_r$ or $C_r$. The vertex range, $V_r$, is $(2:2+\text{bsize})$, while the Center range, $C_r$, is $(2:1+\text{bsize})$.

<table>
<thead>
<tr>
<th>Class</th>
<th>Array shape</th>
<th>Purpose</th>
<th>In-block index ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex</td>
<td>$(ni,ni,ni,ni,ni)$</td>
<td>CV field data</td>
<td>$V_r : V_r : V_r$</td>
</tr>
<tr>
<td>Center</td>
<td>-</td>
<td>CC field data</td>
<td>$C_r : C_r : C_r$</td>
</tr>
<tr>
<td>XiFace</td>
<td>-</td>
<td>$\xi$-face field data</td>
<td>$V_r : C_r : C_r$</td>
</tr>
<tr>
<td>EtaFace</td>
<td>-</td>
<td>$\eta$-face field data</td>
<td>$C_r : V_r : C_r$</td>
</tr>
<tr>
<td>ZetaFace</td>
<td>-</td>
<td>$\zeta$-face field data</td>
<td>$C_r : C_r : V_r$</td>
</tr>
<tr>
<td>XiEdge</td>
<td>-</td>
<td>$\xi$-edge field data</td>
<td>$C_r : V_r : V_r$</td>
</tr>
<tr>
<td>EtaEdge</td>
<td>-</td>
<td>$\eta$-edge field data</td>
<td>$V_r : C_r : V_r$</td>
</tr>
<tr>
<td>ZetaEdge</td>
<td>-</td>
<td>$\zeta$-edge field data</td>
<td>$V_r : V_r : C_r$</td>
</tr>
<tr>
<td>CVsurface</td>
<td>$(ni,ni,6,ni,ni)$</td>
<td>CV, on block surfaces</td>
<td>$V_r : V_r$</td>
</tr>
<tr>
<td>CCsurface</td>
<td>-</td>
<td>CC, across block surfaces</td>
<td>$C_r : C_r$</td>
</tr>
<tr>
<td>CFsurface</td>
<td>-</td>
<td>Faces on block surfaces</td>
<td>$C_r : C_r$</td>
</tr>
<tr>
<td>Cluster</td>
<td>(mcvcl,ncvcl)</td>
<td>Cluster comm. tables</td>
<td>N/A</td>
</tr>
<tr>
<td>Projector</td>
<td>$(ni,ni)$</td>
<td>Simple mask operations</td>
<td>$V_r : V_r$</td>
</tr>
</tbody>
</table>
3.2. MULTIGRID HANDLING

Figure 3.9: Indexing of cell centres and cell faces, relative to the indexing of cell vertices. The cell vertex index range within the block is $(2:\text{bsize}+2)$, whereas the cell center index range within the block is $(2:\text{bsize}+1)$.

treat the coarser levels. In other words, the processors associated with the fine level are idling when the coarser levels are treated, and the execution time for each level turns out to be roughly equal in spite of the great difference in cell numbers.

For this reason, a different array is employed for each multigrid level. In a sense, this leads to a more transparent code. On the other hand, using a maximum of 5 levels, as the present system is prepared for, one has to accept to handle 5 array names in stead of just one. This is a price paid for efficient implementation on architectures as the CM.

It is obvious, that the treatment of many arrays where some represent the same physical data on different multigrid levels, calls for a naming convention. The naming convention adopted here for all data, parameters etc. that coexist on the multigrid levels is the following: the name is constructed of a \textit{generic name} plus a \textit{level indicator}, the latter being a single digit - the number of the multigrid level. The finest level has number 1.

As the crunching routines should be able to handle all levels, the arrays and parameters need to be passed as arguments. This is done by way of a \textit{filter},

```
subroutine example(level)
  implicit none
  include 'params.incl'
  include 'vertices.inc'
  integer level

C---------- Filter: call depends on level ---------------
  if (level.eq.1) then
    call donothing(ni1,bsize1,x1,y1,z1)
  else if (level.eq.2) then
```

CHAPTER 3. IMPLEMENTATION MODEL

```fortran
call donothing(ni2,bsize2,x2,y2,z2)
else if (level.eq.2) then
    call donothing(ni3,bsize3,x3,y3,z3)
else if (level.eq.2) then
    call donothing(ni4,bsize4,x4,y4,z4)
else if (level.eq.2) then
    call donothing(ni5,bsize5,x5,y5,z5)
end if

C------------------------------------------------------------------
return
end

subroutine donothing(ni,bsize,x,y,z)
  implicit none
  include 'params.inc'
  integer ni,bsize
  real*8 x(ni,ni,ni,nb),y(ni,ni,ni,nb),z(ni,ni,ni,nb)
C-This routine does absolutely nothing -----------------
  return
end
```

In the calling routine, a different call is needed for each level. In the called routine, the appended level-indicator is dropped. The coordinate arrays in the example coexist in the calling routine. Here, they are declared in common in the file `vertices.inc`, while the array size parameters are declared in `params.inc`. The fourth array dimension `nb` is the same for each level. It may be transferred by argument or by the include-file as shown. The latter method is necessary for some of the routines in the communication library, since some arrays and array sizes are employed here, that the user should not be bothered with.

The parameter definitions employed in both `Basis3D` and in the applications are given in the include-file

```
PATTERNS.INC

C------------------------------------------------------------------
integer mglev
common/mglevels/mglev
C- Max number of mesh blocks ----------------
integer nb
parameter (nb=10)
C- Array size parameters, level 1 ------
integer ni1,ncvcl1,mcvcl1
parameter( ni1 = 12 , ncvcl1 = 512 , mcvcl1 = 64)
C- Array size parameters, level 2 ------
integer ni2,ncvcl2,mcvcl2
parameter( ni2 = 8 , ncvcl2 = 256 , mcvcl2 = 32)
C- Array size parameters, level 3 ------
integer ni3,ncvcl3,mcvcl3
parameter( ni3 = 6 , ncvcl3 = 128 , mcvcl3 = 16)
C- Array size parameters, level 4 ------
```

3.2. MULTIGRID HANDLING

```c
integer ni4, ncvcl4, mcvcl4
parameter(ni4 = 1, ncvcl4 = 1, mcvcl4 = 1)
```

Array size parameters, level 5

```c
integer ni5, ncvcl5, mcvcl5
parameter(ni5 = 1, ncvcl5 = 1, mcvcl5 = 1)
```

```c
integer nm
parameter(nm = 1000)
```

For reasons of clarity, a lot of comments have been removed here. A copy of all related include-files can be found in appendix A. The other include-file of the above example shows the declaration of a set of coexisting arrays

C VERTICES.INC

```c
real*8 x1, y1, z1, x2, y2, z2, x3, y3, z3, x4, y4, z4, x5, y5, z5
```

```c
common/allcoords/
&x1(ni1, ni1, ni1, nb), y1(ni1, ni1, ni1, nb), z1(ni1, ni1, ni1, nb),
&x2(ni2, ni2, ni2, nb), y2(ni2, ni2, ni2, nb), z2(ni2, ni2, ni2, nb),
&x3(ni3, ni3, ni3, nb), y3(ni3, ni3, ni3, nb), z3(ni3, ni3, ni3, nb),
&x4(ni4, ni4, ni4, nb), y4(ni4, ni4, ni4, nb), z4(ni4, ni4, ni4, nb),
&x5(ni5, ni5, ni5, nb), y5(ni5, ni5, ni5, nb), z5(ni5, ni5, ni5, nb)
```

```c
integer*2 attr1, attr2, attr3, attr4, attr5
```

One advantage of the present transfer model is the possibility of a total separation between of user application and system code, even in cases where the system code is calling user code segments from a great number of different locations.
Chapter 4

Running applications

4.1 Basis3D input

In order to run the Basis3D code, one input file is needed. This file describes the mesh. The contents may be altered during runtime, and is rewritten at the end of the run. In order to describe the proper input format, the routine, reading the input file is shown

```fortran
subroutine readmesh
  implicit none
  include 'params.incl'
  include 'vertices.incl'
  include 'blockdef.incl'
  integer i,j,k,iblock

C-------- Read blocksize and number of blocks -------
  read(2,*)bsize1,nblock

C-----------------------------------------------
  do 6 iblock=1,nblock
C-------------------------- Read block coordinates -------
  do 5 k=2,bsize1+2
  do 5 j=2,bsize1+2
  do 5 i=2,bsize1+2
  5 read(2,*,end=8)attr1(i,j,k,iblock),
    & x1(i,j,k,iblock),
    & y1(i,j,k,iblock),
    & z1(i,j,k,iblock)

C-----------------------------------------------
  6 continue
  return

C---------- Too few data, terminate ---------------
  8 write(6,*)'Too few data in mesh-file'
  stop
  end
```

For reasons of clarity, some comments and the accommodation checks have been removed. After the first input line, giving the number of cells on the finest level and the number of blocks, the vertex data are given, one line per vertex. For each block,
the data for all vertices are given, i.e. vertices on a blockface coinciding with another blockface will appear in different positions of the mesh file.

Each vertex post consists of an attribute and a coordinate triplet. The attribute value reflects the physical significance of the mesh point. Points on a wall have one attribute value, another value signifies an inlet point and so forth, see table 4.1. In order to be able to share mesh data between different applications, the use of special user-defined attributes should be kept to a minimum.

Table 4.1: The attribute values of Basis3D. For points on singular lines, axial/radial directions are listed. Simple BC data are data given by either a single, global value or a global expression.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Physical significance</th>
<th>Assoc. BC data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exterior</td>
<td>0</td>
<td><strong>Vertex outside block</strong></td>
<td></td>
</tr>
<tr>
<td>Interior</td>
<td>1</td>
<td><strong>Interior vertex</strong></td>
<td></td>
</tr>
<tr>
<td>2,3</td>
<td></td>
<td>Singular line, ( \xi/\pm \eta, \pm \zeta )</td>
<td></td>
</tr>
<tr>
<td>4,5</td>
<td></td>
<td>Singular line, ( \eta/ \pm \zeta, \pm \xi )</td>
<td></td>
</tr>
<tr>
<td>6,7</td>
<td></td>
<td>Singular line, ( \zeta/ \pm \xi, \pm \eta )</td>
<td></td>
</tr>
<tr>
<td>Maxinterior</td>
<td>100</td>
<td>Interiors, max attr.</td>
<td></td>
</tr>
<tr>
<td>Wall</td>
<td>101</td>
<td>Wall, adiabatic, noslip</td>
<td>Array-valued</td>
</tr>
<tr>
<td>102-200</td>
<td></td>
<td>Wall, user-specified</td>
<td></td>
</tr>
<tr>
<td>Maxwall</td>
<td>200</td>
<td>Walls, max attr. value</td>
<td></td>
</tr>
<tr>
<td>Inlet</td>
<td>201</td>
<td>Inlet</td>
<td>Simple</td>
</tr>
<tr>
<td>202-300</td>
<td></td>
<td>Inlet, user-specified</td>
<td>Array-valued</td>
</tr>
<tr>
<td>Maxinlet</td>
<td>300</td>
<td>Inlets, max attr. value</td>
<td></td>
</tr>
<tr>
<td>Farfield</td>
<td>301</td>
<td>Farfield</td>
<td>Simple</td>
</tr>
<tr>
<td>302-400</td>
<td></td>
<td>Farfield, user-specified</td>
<td>Array-valued</td>
</tr>
<tr>
<td>Outlet</td>
<td>401</td>
<td>Outlet</td>
<td>Array-valued</td>
</tr>
<tr>
<td>402-500</td>
<td></td>
<td>Outlet, user-specified</td>
<td></td>
</tr>
<tr>
<td>Cyclic</td>
<td>501</td>
<td>Cyclic boundary condition</td>
<td>Array-valued</td>
</tr>
<tr>
<td>502-600</td>
<td></td>
<td>Cyclic BC, attr. value not named</td>
<td></td>
</tr>
<tr>
<td>Symmetry</td>
<td>601</td>
<td>Symmetry plane</td>
<td></td>
</tr>
<tr>
<td>602-700</td>
<td></td>
<td>Symm. plane, attr. not named</td>
<td></td>
</tr>
</tbody>
</table>

The attribute values are checked by the Basis3D. If an exterior point has been given an internal attribute, the point is changed into a wall point. An interior point, that has been given an external attribute is changed to a normal interior point (attribute=1). For coinciding points, if different values are given, the lowest value over the cluster prevails.

When discretizing surfaces, the boundary type will change at some positions. If the change takes place at a vertex, the ambiguity concerning the attribute value needs to be resolved. As a rule of thumb, let the lower value prevail. For example,
if an inlet meets a wall, let the intersecting point be a wall point. Another example is sketched in fig. 4.1, where the stem of a ship hull meets the symmetry plane in a staircase manner. All vertices touching a cell face which is out of the symmetry plane should be wall points, or the hull would be leaking.

All input data are vertex oriented. The attributes corresponding to the cell faces are needed for CC based codes. The values are determined by the Basis3D and stored in an array of the CFsurface class.

![Figure 4.1: Whenever a wall surface intersects another kind of surface, the wall attribute should prevail along the entire intersection.](image)

When running the application, the attribute values are employed to set the boundary condition tables and masks, corresponding to the computational method being used. This is obviously a user job.

The Basis3D code accepts 'unknown' coordinates, within reason. Hence, points on block edges, block faces, or block interiors may be represented by a zero coordinate triplet, i.e. the input line contains a valid attribute value and 3 zeros. This is only valid for entire edges, faces, or interiors. In all cases, the 8 block corners must be represented by their true coordinates. The 'unknown' coordinates will then be calculated by transfinite interpolation.

### 4.2 Basis3D output

Two files of output are generated by the Basis3D. First, the attributes and vertex coordinates are written. Presently, the mesh file connected for input is reused. Then, a file containing the communication tables is written. As a library routine is employed to read the communication file when an application is to start, detailed knowledge of the file contents is not necessary. The arrays written and read can be
seen in the listing of the corresponding declarations in cvtables.inc. The listing appears in appendix A.

4.3 Prototype application

After constructing the mesh file, the Basis3D is run. For the actual mesh, the array sizes may not be adequate. They may be corrected in params.inc. After a correction, Basis3D is recompiled and run. Remember to copy params.inc to the directory, from which you intend to run the subsequent applications.

A number of include-files should be present. As a minimum, the following are needed,

```plaintext
params.inc,
blockdefs.inc,
vertices.inc,
attrib.inc
cvtables.inc,
libspace.inc.
```

A prototype application program might look like fig. 4.2. In this program, the include-files declaring arrays etc. are present in the routines SetMasks and Application. On Unix systems, they must also be present in the main program. Whenever the numerical Algorithm needs to be called, the call is performed through a filter, like shown in the section dealing with Multigrid handling. Hence, Algorithm knows only data associated with one MG level at a time.

Before starting the application itself, the mesh and communication tables are read by one call of Read3D, a routine from the Lib3D library. Eventual user-defined logical masks to be employed by the Algorithm are set by a call of SetMasks, some examples are shown in appendix B.

```
Main Program
  └── Read3D
      └── SetMasks
          └── Application
              └── Filter
                  └── Algorithm

Figure 4.2: Example prototype application program.
4.4 Library routines

In connection to the Basis3D code, a library of routines for communication etc. has been initiated. Presently, the library Lib3D includes some forty-odd code segments. Here, only the calling sequence of those routines, the user is ment to interface with, are presented. The routines are coded in Fortran 77 for sequential architectures. However, versions for parallel or vector machines may be prepared without extreme efforts. A listing of the entire library at its present stage will be made available in the near future. It is noted, that some of the multigrid tools for CC use are not yet fully tested.

The routines are divided in 4 groups, the first group containing the routines employed for file-transfer. The second group takes care of block-block intercommunication. The third group performs the special operations concerned with multigrid techniques. The fourth group contains miscellaneous tools, only used in very special situations.

4.4.1 I/O routines

The present version of the I/O interface is very limited. The present, minimal implementation is concerned only with the above mentioned mesh file and communication file. This is due to the anticipation, that final decisions upon data formats and post-processing facilities will be taken in near future. Following such decisions, the data format may be changed. Also, a more comprehensive and structured use of file transfer for user-defined data will be employed.

Read3D

The routine Read3D does not take any arguments,

\[ \text{call Read3D} \]

It is assumed, that a valid mesh file is connected to unit 2 and a communication file, created by Basis3D to unit 3. Until the above data-formats are decided upon, the open statement are the users responsibility. The Read3D substructure is seen in fig. 4.3. The mesh of the finest level is read by ReadMesh, whereupon the communication tables are read by ReadCom, calling ReadCom1 once for each MG level. Coordinates are copied at coinciding points in order to avoid problems with representation. The coordinates and attributes are injected to the coarser levels.

ReadMesh

The routine ReadMesh is called by Read3D, but may also by called by the user. It takes no arguments,

\[ \text{call ReadMesh} \]

The data format of the mesh file has been discussed in section 4.1. The blocksize on the finest level and the number of blocks is read. The number of multigrid levels that can be accomodated in the arrays defined, is calculated. The number of multigrid levels to be read is determined as the minimum of the accomodatable and
the number of levels written in the communication file. Finally, the attributes and mesh coordinates are read.

**ReadCom**

The routine ReadCom is also called from Read3D, but may be called directly from the user. It takes no arguments,

```fortran
    call ReadCom
```

The communication tables on block level are read. Hereafter, as above mentioned, the pointwise communication tables are read, one MG level at a time.

**WriteMesh**

This code is very similar to ReadMesh, except that it *writes* the mesh file. Like ReadMesh, it takes no arguments.

**WriteCom**

Similar to ReadCom, writes the communication file, blockwise data first, and then the pointwise data, one MG level at a time.

### 4.4.2 Interblock communication routines

On the user level, three different block-block communication routines are at the disposal. Depending on input, they are used in a number of different ways.
Copy3D

During computations, the change of solution in each block needs to be known to the neighbour blocks. The routine Copy3D copies the solution from block to block, for all blocks in the mesh. The calling sequence is

```
call Copy3D(ni,bsize,modus,nvar,var1,var2,var3,var4,var5),
```

where the arguments are

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Shape</th>
<th>Meaning</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>ni</td>
<td>integer*4</td>
<td>1</td>
<td>Array size</td>
<td>-</td>
</tr>
<tr>
<td>bsize</td>
<td>integer*4</td>
<td>1</td>
<td>Blocksize</td>
<td>-</td>
</tr>
<tr>
<td>modus</td>
<td>integer*4</td>
<td>1</td>
<td>Comm. type</td>
<td>[-3:1]</td>
</tr>
<tr>
<td>nvar</td>
<td>integer*4</td>
<td>1</td>
<td># Arrays</td>
<td>[1:5]</td>
</tr>
<tr>
<td>var1-var5</td>
<td>real*8</td>
<td>(ni,ni,ni,nb)</td>
<td>Arrays to copy</td>
<td>-</td>
</tr>
</tbody>
</table>

Between 1 and 5 arrays may be copied within for each call of Copy3D. The number of arrays is given by nvar. The parameter modus determines what kind of copy function to perform. For cell vertex data (CV), modus=1. For cell centered data, modus=0. For the special copy of edge data when calculating face fluxes in cell centered calculations, modus=-1 means that the image data appropriate for calculation of $\xi$-face fluxes are copied in. For $\eta$- and $\zeta$-face fluxes, modus=-2, -3 is used.

Sum3D

In certain vertex centered finite volume discretizations, the residuals are calculated for the part of the control volume inside the block. The residuals at vertices on the block boundaries thereafter are assembled from the residuals at the points in the point clusters or point pairs. This is done by the Sum3D routine. The calling sequence is

```
call Sum3D(ni,bsize,modus,nvar,var1,var2,var3,var4,var5),
```

where the arguments are

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Shape</th>
<th>Meaning</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>ni</td>
<td>integer*4</td>
<td>1</td>
<td>Array size</td>
<td>-</td>
</tr>
<tr>
<td>bsize</td>
<td>integer*4</td>
<td>1</td>
<td>Blocksize</td>
<td>-</td>
</tr>
<tr>
<td>modus</td>
<td>integer*4</td>
<td>1</td>
<td>Comm. type</td>
<td>[-3:1]</td>
</tr>
<tr>
<td>nvar</td>
<td>integer*4</td>
<td>1</td>
<td># Arrays</td>
<td>[1:5]</td>
</tr>
<tr>
<td>var1-var5</td>
<td>real*8</td>
<td>(ni,ni,ni,nb)</td>
<td>Arrays to sum</td>
<td>-</td>
</tr>
</tbody>
</table>

As cell centers can not coincide, the routine only performs for arrays of the Vertex class. Hence, if modus $\neq 1$, nothing is done in the routine.
Sing3D

For finite difference discretizations, the mesh points on singular lines need special treatment. A special molecule might be set up. However, it is a very difficult task to determine a molecule for singular points, that is always in some sense 'good'. Instead, the cluster tables and the attribute list may be employed by the user to generate a molecule which includes differences in all the discrete grid directions at the singular point. Alternatively, in some situations, the solution at the singular point may be determined as the average of the solution in the mesh points one cell away in the radial direction. This technique, which is often employed in mesh generation codes, is implemented in Sing3D. The calling sequence is

```
call Sing3D(ni,bsize,modus,nvar,var1,var2,var3,var4,var5)
```

where the arguments are

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Shape</th>
<th>Meaning</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>ni</td>
<td>integer*4</td>
<td>1</td>
<td>Array size</td>
<td>-</td>
</tr>
<tr>
<td>bsize</td>
<td>integer*4</td>
<td>1</td>
<td>Blocksize</td>
<td>-</td>
</tr>
<tr>
<td>modus</td>
<td>integer*4</td>
<td>1</td>
<td>Comm. type</td>
<td>[-3:1]</td>
</tr>
<tr>
<td>nvar</td>
<td>integer*4</td>
<td>1</td>
<td># Arrays</td>
<td>[1:5]</td>
</tr>
<tr>
<td>var1-var5</td>
<td>real*8</td>
<td>(ni,ni,ni,nb)</td>
<td>Arrays to sum</td>
<td>-</td>
</tr>
</tbody>
</table>

Here, again cell centers can not coincide, and the routine only performs for arrays of the Vertex class.

4.4.3 Multigrid tools

The special interpolation techniques etc. employed with multigrid methods, are represented in Lib3D. Like the block-block communication routines, the multigrid tools are only valid for arrays of the Vertex and Center classes.

Inject3D

When solutions are transferred to coarser multigrid levels, the direct injection is appropriate for PDE's of first and second order. For higher order PDE's, one might get a better performance by using a weighted restriction. As injection is assumed when performing the subsequent prolongations, the user should not endavour to use higher order restriction for solutions, as incorrect behaviour would result!

For CV data, the injection simply copies the values at the points belonging to both fine and coarse mesh from the fine level to the coarse. Nothing is done for the fine mesh points in between the coarse ones. For CC data, the coarse points lie between the fine ones. Hence, the injection is performed as an average of the 8 (in 3D) fine points surrounding the coarse point. The injection is performed by Inject3D. The calling sequence is

```
call Inject3D(nif,nic,bsizec,modus,nvar,
              & varf1,varc1,varf2,varc2,varf3,varc3,
              & varf4,varc4,varf5,varc5)
```
where the arguments are

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Shape</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nif</td>
<td>integer*4</td>
<td>1</td>
<td>Fine array size</td>
</tr>
<tr>
<td>nic</td>
<td>integer*4</td>
<td>1</td>
<td>Coarse array size</td>
</tr>
<tr>
<td>bsizec</td>
<td>integer*4</td>
<td>1</td>
<td>Coarse blocksize</td>
</tr>
<tr>
<td>modus</td>
<td>integer*4</td>
<td>1</td>
<td>Comm. type</td>
</tr>
<tr>
<td>nvar</td>
<td>integer*4</td>
<td>1</td>
<td># Arrays</td>
</tr>
<tr>
<td>varf1-varf5</td>
<td>real*8</td>
<td>(nif,nif,nif,nb)</td>
<td>Fine arrays</td>
</tr>
<tr>
<td>varc1-varc5</td>
<td>real*8</td>
<td>(nic,nic,nic,nb)</td>
<td>Coarse arrays</td>
</tr>
</tbody>
</table>

As above, the argument modus should reflect the array class.

**Restrict3D**

Unlike solutions, residuals are normally transferred to the coarser mesh level by a full weighting. In 3D, full weighting of CV data means, that the residual at a coarse point is determined as a weighted average over the 27 nearest fine-mesh points. For CC residuals, a weighted sum over the 8 nearest mesh points is performed. The calling sequence is similar to the above

```fortran
    call Restrict3D(nif,nic,bsizec,modus,nvar,
                    & varf1,varc1,varf2,varc2,varf3,varc3,
                    & varf4,varc4,varf5,varc5)
```

where the arguments are

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Shape</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nif</td>
<td>integer*4</td>
<td>1</td>
<td>Fine array size</td>
</tr>
<tr>
<td>nic</td>
<td>integer*4</td>
<td>1</td>
<td>Coarse array size</td>
</tr>
<tr>
<td>bsizec</td>
<td>integer*4</td>
<td>1</td>
<td>Coarse blocksize</td>
</tr>
<tr>
<td>modus</td>
<td>integer*4</td>
<td>1</td>
<td>Comm. type</td>
</tr>
<tr>
<td>nvar</td>
<td>integer*4</td>
<td>1</td>
<td># Arrays</td>
</tr>
<tr>
<td>varf1-varf5</td>
<td>real*8</td>
<td>(nif,nif,nif,nb)</td>
<td>Fine arrays</td>
</tr>
<tr>
<td>varc1-varc5</td>
<td>real*8</td>
<td>(nic,nic,nic,nb)</td>
<td>Coarse arrays</td>
</tr>
</tbody>
</table>

**Prolong3D**

The transfer of corrections from a coarse multigrid level to a finer is called a prolongation. Conceptually, prolongations are similar to interpolations. Normally, the prolongations are more complex than the restrictions and injections. When transferring to a finer level for the first time, as in a full multigrid strategy, a high order prolongation is employed, while a simpler version is used, if a correction to a fine-mesh solution is performed.

For CC prolongations, a special problem arises. The correction at the fine points near the block boundary should be interpolated between coarse points inside the
4.4. LIBRARY ROUTINES

block and coarse points outside the block. There exists no rigorous way to establish
the values outside the block, as the boundary conditions are in general not known
to the prolongation routine.

For this reason, for CC discretizations, the data must comply with certain de-
mands.

The calling sequence is

call Prolong3D(nic,nif,bsizec,nact,modus,nvar,
& varc1,varf1,varc2,varf2,varc3,varf3,
& varc4,varf4,varc5,varf5)

where the arguments are

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Shape</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nic</td>
<td>integer*4</td>
<td>1</td>
<td>Coarse array size</td>
</tr>
<tr>
<td>nif</td>
<td>integer*4</td>
<td>1</td>
<td>Fine array size</td>
</tr>
<tr>
<td>bsizec</td>
<td>integer*4</td>
<td>1</td>
<td>Coarse blocksize</td>
</tr>
<tr>
<td>nact</td>
<td>integer*4</td>
<td>1</td>
<td>Choice of action</td>
</tr>
<tr>
<td>modus</td>
<td>integer*4</td>
<td>1</td>
<td>Comm. type</td>
</tr>
<tr>
<td>nvar</td>
<td>integer*4</td>
<td>1</td>
<td># Arrays</td>
</tr>
<tr>
<td>varf1-varf5</td>
<td>real*8</td>
<td>(nif,nif,nif,nb)</td>
<td>Fine arrays</td>
</tr>
<tr>
<td>varc1-varc5</td>
<td>real*8</td>
<td>(nic,nic,nic,nb)</td>
<td>Coarse arrays</td>
</tr>
</tbody>
</table>

It is remarked, that the sequence of coarse and fine data is reversed, relative to the
injection and restriction routines.

One must discriminate between prolongation of solutions and prolongation of
corrections. If a solution exists on the fine level, one should prolongate a correction.
The user signifies his/her choice by the value of nact. The valid combinations of
nact and modus are

<table>
<thead>
<tr>
<th>nact</th>
<th>modus</th>
<th>data type</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;0</td>
<td>1</td>
<td>CV</td>
<td>solution prolongation (c'=c)</td>
</tr>
<tr>
<td>&gt;0</td>
<td>1</td>
<td>CV</td>
<td>correction prolong. (c'=c-I*f)</td>
</tr>
<tr>
<td>+/-1</td>
<td>1</td>
<td>CV</td>
<td>linear prolongation</td>
</tr>
<tr>
<td>+/-3</td>
<td>1</td>
<td>CV</td>
<td>cubic prolongation</td>
</tr>
<tr>
<td>&lt;0</td>
<td>0</td>
<td>CC</td>
<td>solution prolongation (c'=c)</td>
</tr>
<tr>
<td>&gt;0</td>
<td>0</td>
<td>CC</td>
<td>correction prolong. (c'=c-I*f)</td>
</tr>
<tr>
<td>+/-2</td>
<td>0</td>
<td>CC</td>
<td>assume Neuman BC</td>
</tr>
<tr>
<td>-4</td>
<td>0</td>
<td>CC</td>
<td>use coarse exterior data</td>
</tr>
<tr>
<td>+4</td>
<td>0</td>
<td>CC</td>
<td>use fine exterior data</td>
</tr>
</tbody>
</table>

While nact for CV signifies the order of interpolation, for CC it signifies the way
data across block boundaries should be generated.

As above mentioned, for CV data cubic interpolation is employed for prolonga-
tion of solutions, i.e. nact,modus=-3,1, while linear interpolation is employed for
corrections, i.e. \( nact, \text{modus} = 1, 1 \). The other possibilities for CV are normally not used.

For CC data, the way the prolongation is employed depends somewhat on the smoother being used. If a fast smoother is employed, the most effective and simple procedure often is to assume Neuman conditions during the prolongation, \( nact, \text{modus} = 2, 0 \), and subsequently perform a boundary relaxation sweep. For slower smoothers, such as a Runge-Kutta timestep for the Navier-Stokes equations, this is not feasible. Some knowledge about the solution is needed. Hence, if no solution exists on the fine level, \( nact, \text{modus} = -4, 0 \), the user should apply the boundary conditions to extrapolate the solution across the block boundaries. If a solution exists on the fine level, \( nact, \text{modus} = 4, 0 \), this operation should be carried out on the fine level. The extrapolations need not be very precise, so linear extrapolations are quite satisfactory.

### 4.4.4 Miscellaneous tools

If programming advanced tools, in which further knowledge of neighbour blocks than yielded by the above tools is needed, the two below tools may be of interest.

**Newblock**

This routine demands some extra knowledge of the internal structure of a block. The reader is already aware of the numbering of faces on a block. Further, each face has 4 edges and 4 corners. They are numbered 1-4 (of course), in a counterclockwise direction. Corner 1 is in the negative grid directions, while edge 1 connects corners 1 and 2.

Now, given one face and edge of face of a block, the routine first finds the opposite block and blockface. Then, in the opposite block it finds the other face that includes the same edge. This way it is possible to establish the local block topology beyond finding the neighbour blocks across the faces, the blocks across the edges can also be identified, including their orientation.

The calling sequence is

```fortran
call NewBlock(iblock, iface, iedge, icode)
```

where the arguments are

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Shape</th>
<th>Meaning (In)</th>
<th>Meaning (Out)</th>
</tr>
</thead>
<tbody>
<tr>
<td>iblock</td>
<td>integer*4</td>
<td>1</td>
<td>Present block</td>
<td>Opposite block</td>
</tr>
<tr>
<td>iface</td>
<td>integer*4</td>
<td>1</td>
<td>Present face</td>
<td>Outgoing face</td>
</tr>
<tr>
<td>iedge</td>
<td>integer*4</td>
<td>1</td>
<td>Present edge</td>
<td>Outgoing edge</td>
</tr>
<tr>
<td>icode</td>
<td>integer*4</td>
<td>1</td>
<td></td>
<td>Orientation of opposite block</td>
</tr>
</tbody>
</table>

**DecOrien**

The routine name means Decode Orientation. The above routine delivers `icode`, a code for the internal orientation. The present routine translates the code into the
transformation matrix characterizing the orientation of the opposite block, relative to the present. The calling sequence is

\[
\text{call DecOrien(icode,epsilon)},
\]

where the arguments are

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Shape</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>icode</td>
<td>integer*4</td>
<td>1</td>
<td>Orientation of opposite block</td>
</tr>
<tr>
<td>epsilon</td>
<td>integer*4</td>
<td>(3,3)</td>
<td>Unit vectors of opp. block</td>
</tr>
</tbody>
</table>
Chapter 5

Ongoing and contemplated extensions

It is envisaged to make available many of the processes involved in the solution of PDE's as standard tools. Obviously, the relatively stiff (read well-defined) topology concept adopted, broadens the potential range of standardized processes. For example, certain parts of the discretizations in the transformed coordinates may well be of a ready-made nature.

The 'black box' concept is a well-known way of presenting standard tools. However, many numerical methods do not lend themselves to black box treatment. Hence, another concept will be employed for some of the tools to be applied with the herein described structure.

Multigrid techniques employing a Domain Decomposition strategy as smoother for both linear and non-linear systems of equations are an example. A first version of a solver based on these methodologies is in its final development stages. Conceptually, its encapsulation may be described as of a 'grey box' nature, as it in contrast to the black boxes refers to user specified routines. This is done in a simple and straightforward manner, at the same time yielding a potentially great degree of flexibility.

A Navier-Stokes code, based on Runge-Kutta timestepping, is under development. Parts of this project are potential items for standardization, i.e. routines for calculation of metric terms, fluxes, the timestepping scheme etc., may be included in a library.

The I/O sub-system of Basis3D will be updated, when decisions on post-processing software have been taken. Also, a more structured interaction with the operative system should be made possible, i.e. the handling of files associated with many different projects, and different software versions etc. should be automated.

A basic system, similar to the present, for 2D use is contemplated. It is envisaged that this system should not only treat 2D planar topologies, but also be able to handle (closed) surfaces in 3D. Furthermore, it should at some stage be integrated with the present 3D system, via a set of transfer pointers. The potential of such an integration is obvious. Interplay between integral formulations and full Navier-Stokes schemes is made possible, surface grid adaption based on 3D volume solutions is made viable, a simple means of data-reduction is readily at hand, etc.
Acknowledgement

The present work is sponsored through a Post Doctoral Research grant by the Technical University of Denmark.
Bibliography


Appendix A

The Basis3D include files

The include-files employed in the topology file generator and in the user codes are presented. To be employed, they are copied to the users own directory, and changed to fit the problem being solved. Probably only one file, params.inc, needs changes. The user-copied include-files are employed both during compilation of the topology file generator and in the users own code. In this way the topology file generator is tailored to the user-problem. The include-files employed in the Basis3D code are listed in table A.1. Below, the entire include-files are printed, with all comments, etc. It is emphasized, that the filenames are all in lower-case, this is important for most operating systems.

Table A.1: The include files used in the Basis3D code. Also shown is the scope and contents of each file. None of the include files used in applications can be employed unless params.inc is also included.

<table>
<thead>
<tr>
<th>File name</th>
<th>Scope</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Basis3D</td>
<td>Applications</td>
</tr>
<tr>
<td>params.inc</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>attrib.inc</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>blockcon.inc</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>blockdef.inc</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>cvtables.inc</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>cvmasks.inc</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>decode.inc</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>libspace.inc</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>vertices.inc</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

A.1 Params.inc

C================================= number of Multi-gridlevels =========
c

A-1
APPENDIX A. THE BASIS3D INCLUDE FILES

C OBS! as MGLEV is determined by comparing the actual meshsize
to the (herein) preset array-sizes for each level, there
is no parameter statement for MGLEV.

C------------------------------------------------------------------
integer mglev
common/mglevels/mglev
C=========================== Max number of mesh blocks ==========
integer nb
parameter (nb=10)
C=========================== Array size parameters for each MG level =
C Parameters are named NAMElevel, where NAME is the generic name,
C level is the MG-level. Thus,
C
NI1 is max mesh-block size for level 1
The cell vertices defining a mesh-block are in the range
(2:2+BSIZE , 2:2+BSIZE , 2:2+BSIZE ), and as one
extra location is needed in each +/- index direction, NI
should exceed the number of cells, BSIZE, by at least 3.
C
NCVCL1 is max number of Cell-Vertex CLusters on level 1
MCVCL1 is max size of Cell-Vertex CLusters on level 1
(Coincident meshpoints on block edges/corners are register
as vertex clusters by the Basis3D preprocessor)

C--------------------------------- Array size parameters, level 1 -
integer ni1,ncvcl1,mcvcl1
parameter( ni1 = 12 , ncvcl1 = 512 , mcvcl1 = 64)
C--------------------------------- Array size parameters, level 2 -
integer ni2,ncvcl2,mcvcl2
parameter( ni2 = 8 , ncvcl2 = 256 , mcvcl2 = 32)
C--------------------------------- Array size parameters, level 3 -
integer ni3,ncvcl3,mcvcl3
parameter( ni3 = 6 , ncvcl3 = 128 , mcvcl3 = 16)
C--------------------------------- Array size parameters, level 4 -
integer ni4,ncvcl4,mcvcl4
parameter( ni4 = 1 , ncvcl4 = 1 , mcvcl4 = 1)
C--------------------------------- Array size parameters, level 5 -
integer ni5,ncvcl5,mcvcl5
parameter( ni5 = 1 , ncvcl5 = 1 , mcvcl5 = 1)
C=========================== Auxillary parameters ==========
C
C Parameters employed in solvers etc.
C
C-------- DoDec3D --- nm > (ni1**2+ni2**2+ni3**2+ni4**2+ni5**2)*4 -
integer nm
The pre-defined attribute values. The ranges of non-defined attributes are reserved as follows.

<table>
<thead>
<tr>
<th>Name</th>
<th>Range/Value</th>
<th>Definition</th>
<th>Reservation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exterior</td>
<td>0</td>
<td>Vertex outside block</td>
<td>-</td>
</tr>
<tr>
<td>Interior</td>
<td>1</td>
<td>Normal interior vertex</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>8 - 50</td>
<td></td>
<td>Reserved for grid generation.</td>
</tr>
<tr>
<td></td>
<td>51 - 100</td>
<td></td>
<td>Spec. interiors.</td>
</tr>
<tr>
<td>MaxInterior</td>
<td>100</td>
<td>Highest interior attr.</td>
<td>-</td>
</tr>
<tr>
<td>Wall</td>
<td>101</td>
<td>Wall, noslip, adiabatic, impermeable</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>102-200</td>
<td>Userspec. walls</td>
<td>-</td>
</tr>
<tr>
<td>MaxWall</td>
<td>200</td>
<td>Highest wall attr.</td>
<td>-</td>
</tr>
<tr>
<td>Inlet</td>
<td>201</td>
<td>Constant value (or function) inlet</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>202-300</td>
<td>Array-valued inlets</td>
<td>(user-spec)</td>
</tr>
<tr>
<td>MaxInlet</td>
<td>300</td>
<td>Highest inlet attr.</td>
<td>-</td>
</tr>
<tr>
<td>Farfield</td>
<td>301</td>
<td>Farfield, U=const (+circulation)</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>302-400</td>
<td>Farfield, userspec.</td>
<td>-</td>
</tr>
<tr>
<td>Outlet</td>
<td>401</td>
<td>Outlet, non-valued</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>402-500</td>
<td>Outlet, userspec.</td>
<td>-</td>
</tr>
<tr>
<td>Cyclic</td>
<td>501</td>
<td>Cyclic BC</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>502-600</td>
<td>(attr. not named)</td>
<td>-</td>
</tr>
<tr>
<td>Symmetry</td>
<td>601</td>
<td>Symmetry-plane</td>
<td>-</td>
</tr>
</tbody>
</table>

integer exterior,
& interior, maxinterior,
& wall , maxwall,
& inlet , maxinlet,
& outlet ,
& farfield,
& cyclic,
& symmetry
parameter (exterior= 0,
& interior= 1, maxinterior=100,
& wall = 101, maxwall = 200,
& inlet =201,maxinlet =300,
& farfield=301,
& outlet =401,
& cyclic =501,
& symmetry=601)

A.2 Atrib.inc

C------------------------------------------------------------------
C Declaration of CV and CC attribute arrays, all MG levels
C------------------------------------------------------------------

integer*2 attrla,attr2,attr3,attr4,attr5
integer*2 attrcc1,attrcc2,attrcc3,attrcc4,attrcc5

C==================================================================
Like the names employed in params.inc the names here consist of a generic name with a digit appended, corresponding to the multigrid level on which the array is to be used. Hence, the above file declares one array for each MG level.

A.3 Blockcon.inc

C====== nbc => nb, ncl: approximately 3nb, nedge: appr. 12nb =====

integer nbcc,ncl,nedge,ncorn
parameter (nbcc=500,ncl=1500,nedge=5000,ncorn=5000)
C==================== Max distance between 'coincident' points ====
real*8 dxmax

C================================ Tables of edge-clusters ======

integer edgepnt,edgefirst,edgelast
integer edgeblck,edgestart,edgeend,nedgecl

C================================ Tables of corner-clusters ====
### A.4 Blockdef.inc

**Size and number of blocks**

```fortran
integer &nbblock,bsize1,bsize2,bsize3,bsize4,bsize5
common/bdefs/
```

**Tables of face-face connections**

```fortran
integer fnbblc,fnbfac,fnbori,fnbdir
common/faceb/fnbblc(6,nb),fnbfac(6,nb),
& fnbori(6,nb),fnbdir(6,nb)
```

**Format for mesh output**

```fortran
character*31 formtext
common/meshform/formtext
```

The computational blocksizes on the different MG levels are designated by `bsize1` through `bsize5`, while `nbblock` is the number of computational blocks. The arrays `fnbblc`, `fnbfac`, `fnbori` and `fnbdir` representing the topology on block level, are probably of little relevance to the user.

The character variable `formtext` contains the I/O format to be employed for writing coordinate data.

### A.5 Cvtables.inc

**Declarations for level 1**

```fortran
integer clmax1
integer*2 clblcl,cllcl,clj1,clk1,
& imacv1,jmacv1,kmacv1,bmacv1,cltyp1,clpnt1
real*8 collect1,sigma1
logical sumcv1,copcv1
integer*2 imacc1,jmaccl,kmaccl,bmacc1
logical copcc1
```

**Tables for point clusters, level 1**

```fortran
common/cllevi/clmax1
common/cllevib/clblcl(mvcvcl1,ncvcv1),cll1 (mcvcv1,ncvcv1),
```
& clj1 (mcvcl1,ncvcl1),clk1 (mcvcl1,ncvcl1),
& imacv1(ni1,ni1,ni1,nb),jmacv1(ni1,ni1,ni1,nb),
& kmacv1(ni1,ni1,ni1,nb),bmacv1(ni1,ni1,ni1,nb),
& cltyp1(ncvcl1)
common/cllevic/collect1(mcvcl1,ncvcl1),sigma1(ncvcl1)
common/cllevid/sumcv1(ni1,ni1,ni1,nb),copcv1(ni1,ni1,ni1,nb)
common/cllevie/clptnt1(ni1,ni1,ni1,nb)
common/cllevif/imacc1(ni1,ni1,6,nb),jmacc1(ni1,ni1,6,nb),
& kmacc1(ni1,ni1,6,nb),bmacc1(ni1,ni1,6,nb)
common/cllevig/copcc1(ni1,ni1,6,nb)

C============================================
integer clmax2
integer*2 clblc2,cli2,clj2,clk2,
& imacv2,jmacv2,kmacv2,bmacv2,cltyp2,clptn2
real*8 collect2,sigma2
logical sumcv2,copcv2
integer*2 imacc2,jmacc2,hacc2,bmacc2
logical copcc2

C============================================ Tables for point clusters, level 2 ===
common/cllev2a/clmax2
common/cllev2b/clblc2(mcvcl2,ncvcl2),cli2 (mcvcl2,acvcl2),
& clj2 (mcvcl2,ncvcl2),clk2 (mcvcl2,ncvcl2),
& imacv2(ni2,ni2,ni2,nb),jmacv2(ni2,ni2,ni2,nb),
& kmacv2(ni2,ni2,ni2,nb),bmacv2(ni2,ni2,ni2,nb),
& cltyp2(ncvcl2)
common/cllev2c/collect2(mcvcl2,ncvcl2),sigma2(ncvcl2)
common/cllev2d/sumcv2(ni2,ni2,ni2,nb),copcv2(ni2,ni2,ni2,nb)
common/cllev2e/clptn2(ni2,ni2,ni2,nb)
common/cllev2f/imacc2(ni2,ni2,6,nb),jmacc2(ni2,ni2,6,nb),
& kmacc2(ni2,ni2,6,nb),bmacc2(ni2,ni2,6,nb)
common/cllev2g/copcc2(ni2,ni2,6,nb)

C============================================
integer clmax3
integer*2 clblc3,cli3,clj3,clk3,
& imacv3,jmacv3,kmacv3,bmacv3,cltyp3,clptn3
real*8 collect3,sigma3
logical sumcv3,copcv3
integer*2 imacc3,jmacc3,kmacc3,bmacc3
logical copcc3

C=========================================== Tables for point clusters, level 3 ===
common/cllev3a/clmax3
common/cllev3b/clblc3(mcvcl3,ncvcl3),cli3 (mcvcl3,ncvcl3),
& clj3 (mcvcl3,ncvcl3),clk3 (mcvcl3,ncvcl3),
& imacv3(ni3,ni3,ni3,nb),jmacv3(ni3,ni3,ni3,nb),
& kmacv3(ni3,ni3,ni3,nb),bmacv3(ni3,ni3,ni3,nb),
& cltyp3(ncvcl3)
common/cllev3c/collect3(mcvcl3,ncvcl3),sigma3(ncvcl3)
Declarations for level 4 ========

```
integer clmax4
integer*2 clblc4,cli4,clj4,clk4,
  & imacv4,jmacv4,kmacv4,bmacv4,cltyp4,clpnt4
real*8 collect4,sigma4
logical sumcv4,copcv4
integer*2 imacc4,jmacc4,kmacc4,bmacc4
logical copcc4
```

Tables for point clusters, level 4 ======

```
common/cllev4a/clmax4
common/cllev4b/clblc4(mcvcl4,ncvcl4),cli4 (mcvcl4,ncvcl4),
  & clj4 (mcvcl4,ncvcl4),clk4 (mcvcl4,ncvcl4),
  & imacv4(ni4,ni4,ni4,nb),jmacv4(ni4,ni4,ni4,nb),
  & kmacv4(ni4,ni4,ni4,nb),bmacv4(ni4,ni4,ni4,nb),
  & cltyp4(ncvcl4)
common/cllev4c/collect4(mcvcl4,ncvcl4),sigma4(ncvcl4)
common/cllev4d/sumcv4(ni4,ni4,ni4,nb),copcv4(ni4,ni4,ni4,nb)
common/cllev4e/clpnt4(ni4,ni4,ni4,nb)
common/cllev4f/imacc4(ni4,ni4,6,nb),jmacc4(ni4,ni4,6,nb),
  & kmacc4(ni4,ni4,6,nb),bmacc4(ni4,ni4,6,nb)
common/cllev4g/copcc4(ni4,ni4,6,nb)
```

Declarations for level 5 ========

```
integer clmax5
integer*2 clblc5,cli5,clj5,clk5,
  & imacv5,jmacv5,kmacv5,bmacv5,cltyp5,clpnt5
real*8 collect5,sigma5
logical sumcv5,copcv5
integer*2 imacc5,jmacc5,kmacc5,bmacc5
logical copcc5
```

Tables for point clusters, level 5 ======

```
common/cllev5a/clmax5
common/cllev5b/clblc5(mcvcl5,ncvcl5),cli5 (mcvcl5,ncvcl5),
  & clj5 (mcvcl5,ncvcl5),clk5 (mcvcl5,ncvcl5),
  & imacv5(ni5,ni5,ni5,nb),jmacv5(ni5,ni5,ni5,nb),
  & kmacv5(ni5,ni5,ni5,nb),bmacv5(ni5,ni5,ni5,nb),
  & cltyp5(ncvcl5)
common/cllev5c/collect5(mcvcl5,ncvcl5),sigma5(ncvcl5)
common/cllev5d/sumcv5(ni5,ni5,ni5,nb),copcv5(ni5,ni5,ni5,nb)
common/cllev5e/clpnt5(ni5,ni5,ni5,nb)
common/cllev5f/imacc5(ni5,ni5,6,nb),jmacc5(ni5,ni5,6,nb),
  & kmacc5(ni5,ni5,6,nb),bmacc5(ni5,ni5,6,nb)
common/cllev5g/copcc5(ni5,ni5,6,nb)
```
The above arrays contain information on the cell vertices. The scalar \texttt{clmaxl} is the number of point clusters on MG level 1. The arrays \texttt{clil}, \texttt{clj1}, \texttt{clk1}, \texttt{clblc1} hold the indices \((i, j, k, iblock)\) of the points in the clusters. First index in those arrays represents the point number in a cluster, while second index represents the cluster number. The array \texttt{cltyp1} contains the number of points in each cluster. The number may be negative, thus designating a cluster on a singular line. The arrays \texttt{imacv1}, \texttt{sigma1} are empty, they will be employed as workspace by communication routines on parallel architectures.

The arrays \texttt{imacv1}, \texttt{jmacv1}, \texttt{kmacv1}, \texttt{bmacv1} hold the indices of point pairs. For each mesh point \((i, j, k, n)\) (on a blockface), the indices of its (eventual) imagepoint is stored in the \((i, j, k, n)\)-position of the 4 above arrays.

The logical array \texttt{copcv1} is a mask employed when information is copied from block to block. At locations \((i, j, k, n)\) where it is \(\text{true.}\), a copy of the contents (of the same array) at the position given by the arrays \texttt{imacv1}, \texttt{jmacv1}, \texttt{kmacv1}, \texttt{bmacv1} is performed. The array \texttt{sumcv1} functions in much the same way when a sum is needed.

The arrays employed for communication of cell centered data have a \texttt{cc} in their name instead of the \texttt{cv}, i.e. \texttt{imacc1} etc.

### A.6 Cvmasks.inc

```c
C====================================== Workspace masks used in Basis3D ======
   logical cvmask1,cvmask2,cvmask3,cvmask4,cvmask5
   common/cvmasks/cvmask1(nil,ni1,nil,nb),
   &      cvmask2(ni2,nil2,nil2,nb),
   &      cvmask3(nil3,nil3,nil3,nb),
   &      cvmask4(ni4,nil4,nil4,nb),
   &      cvmask5(nil5,nil5,nil5,nb)
C======================================
```

### A.7 Decode.inc

```c
C====================================== Arrays for decoding wireframe structure =====
C The arrays are initiated in BLOCK DATA
C======================================
   integer faceface,faceedge,edgelogl,ec,edgeface,ffedge
   integer facedef,edgedef,cornedge,corn3d,
   &      right3d,cornadj,face3d
   common/decode1/faceface(4,6),faceedge(12,6),edgelogl(4,6)
   common/decode2/ec(8),edgeface(2,12),ffedge(6,6)
   common/decode3/facedef(4,6),edgedef(2,12),cornedge(3,8)
   common/decode4/corn3d(3,8),
   &      right3d(8,8),cornadj(3,8),face3d(8,8)
C======================================
```
A.8 Libspace.inc

C======================================================== Workspace used in Lib3D ======
                  real*8 wp011, wp012, wp013, wp014
          common/wpsp1/wp011(ni1, ni1, ni1, ni1, nb), & wp012(ni2, ni2, ni2, nb), & wp013(ni3, ni3, ni3, nb), & wp014(ni4, ni4, ni4, nb)
C=========================================================

A.9 Vertices.inc

C======================================================== Definitions of coordinates, all levels ======
                  real*8 x1, y1, z1, x2, y2, z2, x3, y3, z3, x4, y4, z4, x5, y5, z5
C========================================================
                  common/allcoords/
                   &x1(ni1, ni1, ni1, nb), y1(ni1, ni1, ni1, nb), z1(ni1, ni1, ni1, nb),
                   &x2(ni2, ni2, ni2, nb), y2(ni2, ni2, ni2, nb), z2(ni2, ni2, ni2, nb),
                   &x3(ni3, ni3, ni3, nb), y3(ni3, ni3, ni3, nb), z3(ni3, ni3, ni3, nb),
                   &x4(ni4, ni4, ni4, nb), y4(ni4, ni4, ni4, nb), z4(ni4, ni4, ni4, nb),
                   &x5(ni5, ni5, ni5, nb), y5(ni5, ni5, ni5, nb), z5(ni5, ni5, ni5, nb)
C=========================================================
Appendix B

Setting up logical masks

In order to facilitate the coding of application programs, the use of logical masks may be of some interest. For codes written in Fortran 90, the masks enable the use of `where - elsewhere - endwhere` constructs instead of more complicated codes. Even for codes written in Fortran 77, the masks often can make the codes both more efficient and more transparent. Based upon the attribute arrays and the communication tables produced by Basis3D, the user has an array of different ways to produce masks. In most cases, the masks are needed for Finite difference and cell-vertex based Finite volume use. For cell-centered use, the attribute arrays etc. may with reason be employed directly. Some examples will be shown here.

B.1 Finite differences with Dirichlet BC

In the present example, the solution values are assumed at the boundaries. Hence, the user needs to distinguish between boundary points and interior points. Moreover, since the difference molecules created for the different point belonging to a point cluster will differ, except when the cluster is internal and contains 4 points. The mask is named FDmask and should be `.TRUE.` for the points, where the user will perform calculations. A different mask is made for each multigrid level, the `full` name of the mask for level 1 thus becomes FDmask1.

```fortran
subroutine setmasks
implicit none
#include 'params.inc'
#include 'blockdef.inc'
#include 'cvtables.inc'
#include 'vertices.inc'
#include 'attrib.inc'
#include 'usermask.inc'
if(mglev.ge.1) call setmask(ni1,bsize1,nblock,clmax1,
  &  mcvcl1,ncvcl1,attr1,copcv1,
  &  cltyp1,cli1,clj1,clk1,clblc1,
  &  fdmask1)
if(mglev.ge.2) call setmask(ni2,bsize2,nblock,clmax2,
  &  mcvcl2,ncvcl2,attr2,copcv2,
  &  cltyp2,cli2,clj2,clk2,clblc2,
```
APPENDIX B. SETTING UP LOGICAL MASKS

if(mglev.ge.3) call setmask(ni3,bsize3,nblock,clmax3, & mcvcl3,ncvcl3,attr3,copcv3, & cltyp3,cli3,clj3,clk3,clblc3, & fdmask3)
if(mglev.ge.4) call setmask(ni4,bsize4,nblock,clmax4, & mcvcl4,ncvcl4,attr4,copcv4, & cltyp4,cli4,clj4,clk4,clblc4, & fdmask4)
if(mglev.ge.5) call setmask(ni5,bsize5,nblock,clmax5, & mcvcl5,ncvcl5,attr5,copcv5, & cltyp5,cli5,clj5,clk5,clblc5, & fdmask5)
return
end

subroutine setmask(ni,bsize,nblock,clmax, & mcvcl,ncvcl,attr,copcv, & cltyp,cli,clj,clk,clblc, & fdmask)
implicit none
include 'params.inc'
integer ni,bsize,nblock,clmax,mcvcl,ncvcl,i,j,k,n,icl,m,ncl
integer*2 attr(ni,ni,ni,ni)
integer*2 cltyp(ncvcl),cli(mcvcl,ncvcl),clj(mcvcl,ncvcl), & clk(mcvcl,ncvcl),clblc(mcvcl,ncvcl)
logical copcv(ni,ni,ni,ni)
logical fdmask(ni,ni,ni,ni)

C-------------------------  Scan all points in mesh --------------------------
do 1 n=1,nblock
do 1 k=2,bsize+2
do 1 j=2,bsize+2
do 1 i=2,bsize+2
C----- Set all interiors .TRUE. and all boundary .FALSE. ----------
if(attr(i,j,k,n).le.maxinterior) then
  fdmask(i,j,k,n)=.true.
else
  fdmask(i,j,k,n)=.false.
end if
1 continue
C----- Correct clusters, so incorrect molecules are not used ------
do 3 m=1,clmax
  ncl=abs(cltyp(m))
  if(ncl.ne.4) then
    do 2 icl=2,ncl
      fdmask(cli (icl,m), & clj (icl,m), &
B.2. CELLS TOUCHING AN EXTERIOR FACE

It is noted, that the mask does not distinguish between ordinary and singular points. If the singular point should be excluded, the body of loop 1 should read

```plaintext
... if(attr(i,j,k,n).eq.interior) then
    fdmask(i,j,k,n)=.true.
else
    fdmask(i,j,k,n)=.false.
end if
...
```

In the example, the masks are declared in the include-file `usermask.incl`:

```plaintext
C================= Definitions of coordinates, all levels ========
logical fdmask1,fdmask2,fdmask3,fdmask4,fdmask5
C===============================================
common/allmask/fdmask1(ni1,ni1,ni1,nb),
& fdmask2(ni2,ni2,ni2,nb),
& fdmask3(ni3,ni3,ni3,nb),
& fdmask4(ni4,ni4,ni4,nb),
& fdmask5(ni5,ni5,ni5,nb)
C===============================================
```

B.2 Cells touching an exterior face

The present mask is intended for cell-centered use. All cells that touch an exterior cell-face, are marked .TRUE., and are in the user code fixed up according to their specific boundary condition. This example also conveniently illustrates the connection between cell-face and cell-center indexing.

```plaintext
subroutine setmask(ni,bsize,nblock,attrcc,fvmask)
implicit none
include 'params.inc'
integer ni,bsize,nblock,i,j,n
integer*2 attrc(ni,ni,6,nb)
logical fvmask(ni,ni,nb)
C--------------- Initiate mask for all interior cells --------
    do 1 n=1,nblock
    do 1 k=2,bsize+1
    do 1 j=2,bsize+1
    do 1 i=2,bsize+1
```

```plaintext
& clk (icl,m),
& clblc(icl,m)=.false.
2 continue
    end if
3 continue
    return
end
```

```plaintext
C------------------ Initiate mask for all interior cells ---------
    do 1 n=1,nblock
    do 1 k=2,bsize+1
    do 1 j=2,bsize+1
    do 1 i=2,bsize+1
```
fvmask(i,j,k,n) = .false.

1 continue

C------------------ Scan the cell-face attribute array ------------

do 2 n=1,nblock
    do 2 i=2,bsize+1
      do 2 j=2,bsize+1
        if(attrcc(i,j,1,n).gt.maxinterior)
          fvmask(2,i,j,n) = .true.
        &
        if(attrcc(i,j,2,n).gt.maxinterior)
          fvmask(bsize+1,i,j,n) = .true.
        &
        if(attrcc(i,j,3,n).gt.maxinterior)
          fvmask(j,2,i,n) = .true.
        &
        if(attrcc(i,j,4,n).gt.maxinterior)
          fvmask(j,bsize+1,i,n) = .true.
        &
        if(attrcc(i,j,5,n).gt.maxinterior)
          fvmask(i,j,2,n) = .true.
        &
        if(attrcc(i,j,6,n).gt.maxinterior)
          fvmask(i,j,bsize+1,n) = .true.
    2 continue

return

end