Publ. Nr 93/3

CHALMERS TEKNISKA HÖGSKOLA
Institutionen för Termo- och Fluiddynamik

CHALMERS UNIVERSITY OF TECHNOLOGY
Department of Thermo- and Fluid Dynamics

A Multiblock-Moving Mesh Extension to the CALC-BFC Code

by

Thomas Hellström and Lars Davidson

Thermo and Fluid Dynamics
Chalmers University of Technology
412 96 Gothenburg, Sweden

Göteborg, Januari 1993

Contents

1 Introduction **3**

I User's guide **3**

2 Definitions **3**
2.1 General . . . 3
2.1.1 Computational domain . . . 3
2.1.2 Boundary treatment . . . 4
2.1.3 Moving mesh . . . 4
2.2 Blocks and faces . . . 4
2.2.1 indexing of blocks and numbering of faces . . . 4
2.2.2 indexing of faces . . . 4
2.2.3 Specifying a window . . . 5

3 Specifying the computational domain **5**
3.1 Compile-time parameters . . . 5
3.2 Running time initial settings . . . 5
3.2.1 overview . . . 5
3.2.2 The subroutine 'initfa' . . . 6
3.2.3 The subroutine 'snowin' . . . 6
3.2.4 The subroutine 'setwin' . . . 6
3.2.5 Action.f - The boundary type library . . . 7
3.2.6 The user specified routines 'setbou' and 'setio' . . . 9
3.2.7 Correcting the outflow convection . . . 12
3.2.8 Specifying the pressure reference node . . . 12
3.2.9 Optimizing for two-dimensional calculations . . . 12
3.2.10 Specifying the number of ghost cell layers used . . . 13
3.2.11 Specifying the time discretization scheme. . . . 13
3.2.12 Initializing the flowfield . . . 13
3.2.13 Initializing the turbulent quantities . . . 13
3.2.14 Selecting the blocks to use . . . 14
3.2.15 Setting convergence criterion . . . 14
3.2.16 Debugging block connectivity . . . 14
3.2.17 Specifying constants for Rhie & Chow interpolation . . . 14

4 Examples **14**
4.1 Specifying setup data with subroutine calls . . . 14
4.2 Reading setup data from data files . . . 16

5 Moving mesh specification **19**

II Technical description **20**

6 Brief multiblock subroutine description **20**
6.1 Selecting a block . . . 20
6.2 Data types . . . 20
6.3 The routine 'store' - store variable layer . . . 21
6.4 The routine 'sind' - store index layer . . . 21
6.5 The routine 'facret' - Transfer ghost cell buffer back to block . . . 22

6.6 The ghost cell value extrapolation routines . . . 22
6.6.1 The routine ‘tface’ - Build ghost cell values from a remote face layer buffer . . . 22
6.6.2 The routine ‘xtrapl’ - Transfer values from a layer buffer to a ghost cell buffer . . . 24
6.6.3 The routine ‘setval’ . . . 24
6.6.4 The routine ‘ioval’ . . . 24
6.6.5 The routine ‘xxtra’ . . . 25
6.7 The window information database ‘winbas.f’ . . . 25
6.8 The layer buffer data base - ‘base.f’ . . . 26
6.9 The general ghost cell building routines . . . 28

7 The geometry of the ghost cells **28**
7.1 Required geometrical quantities . . . 28
7.1.1 Treatment at multiblock boundaries . . . 29
7.1.2 Treatment at domain boundaries . . . 29

8 Modifications to standard routines **30**
8.1 The routine ‘wallf’ - Set wall functions . . . 30
8.2 The routine ‘mdcon’ - compute *conv* at domain boundaries . . . 30
8.3 The global convection correction routines ‘conset’ and ‘upconv’ . . . 30
8.4 ‘Upcoef’ - Zero domain boundary coefficients . . . 31
8.5 ‘Ucoef’ - Zero outflow coefficients . . . 31
8.6 Splitting of the routine ‘calcp’ . . . 31
8.7 Modifications to ‘Coeff’ . . . 31
8.8 Modifications to ‘Update’ . . . 31

9 Moving mesh routines **31**

10 Theoretical explanation of some implementations **32**
10.1 The QUICK scheme . . . 32
10.2 Adaptive under-relaxation . . . 33
10.3 The second-order BDF time-discretization scheme . . . 36

1 Introduction

This report presents an extension to the CALC-BFC code which makes it possible to divide the computational domain into several blocks and to make use of meshes that moves in time. It is assumed that the reader is already familiar with the CALC-BFC code.

Part I
User’s guide

2 Definitions

2.1 General

2.1.1 Computational domain

In the present code the calculation domain is divided into *blocks*, and each block has six *faces*. Each face is in turn divided into *windows*. The maximum number of blocks in the computational domain, as well as the maximum number of windows on a face may be specified in the file ‘common.for’. For each window,

a boundary type may be specified. In particular it is possible to match a window of a face to any other window specified, as long as the mesh at the two windows' location coincides.

2.1.2 Boundary treatment

Each block is automatically supplied with two layers of ghost cells. The values of these ghost cells are computed so that, by use of linear interpolation, the value at the block boundary coincides with the desired one. At a multiblock boundary, the values of a ghost cell is simply set to the value of the corresponding cell in the neighboring block.

2.1.3 Moving mesh

If a solution to a time-dependent problem is to be computed, it is possible to specify the current velocity of each *cell face*, and the current location of each grid point, and in this way compute a solution on a moving mesh. This facility is not suitable for Lagrangian computations, but should be seen as an aid to adapt the mesh to a computational domain, the dimensions of which change in time, for example a cylinder with a moving piston.

2.2 Blocks and faces

2.2.1 indexing of blocks and numbering of faces

The cells in each block is numbered, as in standard CALC using the indices i,j and k, and the faces are defined as follows:

- East face, corresponding to the highest i index, face number 1.
- North face, corresponding to the highest j index, face number 2.
- West face, corresponding to the lowest i index, face number 3.
- South face, corresponding to the lowest j index, face number 4.
- High face, corresponding to the highest k index, face number 5.
- Low face, corresponding to the lowest k index, face number 6.

The faces 'West', 'South' and 'Low' are *negative* faces and the other faces are *positive*.
For the negative faces, the outer ghost cell has *dependent* index 0, and the first cell in the computational domain has *dependent* index 2. For the west face, for example, the outer ghost cell has i index 0, and thus the dependent index is the i index.

2.2.2 indexing of faces

Since the faces are two-dimensional, we need only two indices to specify a window. These indices are called the m and n indices and can be obtained by omitting the dependent index, which is constant on a face in the following way:

- For face number one, or the east face, the m index is index j and the n index is index k.
- For face number two, or the north face, the m index is index i and the n index is index k.
- For face number three, or the west face, the m index is index j and the n index is index k.
- For face number four, or the south face, the m index is index i and the n index is index k.
- For face number five, or the high face, the m index is index i and the n index is index j.

- For face number six, or the low face, the m index is index i and the n index is index j.

One simple way to remember this rule is to determine which two indices are *independent* for the particular face and order them in alphabetic order. For the high face, for example, the dependent index is index k, and the independent indices are i and j, ordered alphabetically. Thus the m index corresponds to index i and the n index to index j.

2.2.3 Specifying a window

A window is always rectangular and is specified by the lower left and upper right corners in the m,n coordinate system. If we, for example want to specify a window on the east face, ranging from $k = 2$ to $k = 6$ and from $j = 5$ to $j = 10$, we first have to order the indices in alphabetic order to obtain the m, n coordinate system. Following the above rule we see that index m equals index j and index n equals index k. The lower left corner in the m, n system thus becomes $(5, 2)$ and the upper right corner $(10, 6)$.

3 Specifying the computational domain

3.1 Compile-time parameters

First of all, the file 'common.for' has to be adjusted to be able to cope with the amount of data being specified. The parameters that need to be specified are

- it = Max number of cells in i direction + 4.
- jt = Max number of cells in j direction + 4.
- kt = Max number of cells in k direction + 4.
- $msiz$ = Number of cells in the longest block face = $\max it, jt, kt$.
- $maxbl$ = Max number of blocks.
- $ifsiz$ = Total number of ghost cells / 2. The program stops and warns if this parameter is too small.
- $iomax$ = Total number of cells in the whole domain including ghost cells. The program stops and warns if this parameter is too small.
- $imwin$ = Max number of windows on a block face.

3.2 Running time initial settings

3.2.1 overview

When this is done, the subroutine 'setup' should do the following:

- Set the variable *ngrid* to the number of blocks used.
- The logical variable *twodim* should be set to it's appropriate value, see below.
- The logical variable *mmesh* should be set to the appropriate value, see section 5.
- Before the mesh is read in for each block, the subroutine 'initfa' should be called with the dimensions of the block. The subroutine 'key' will then be able to address the new block. It is the routine 'initfa' that gives the warnings if the parameters $ifsiz$ and $iomax$ are too small.
- Read the mesh.

- Specify the block connectivity and the boundary conditions using the subroutines ‘snowin’ and ‘setwin’.
- Do the usual setup.

It is possible to add a block using ‘initfa’ and to redefine the block connectivity and the boundary conditions using ‘snowin’ and ‘setwin’ when the computations have already started. It is, however, not possible to remove a block at running time.

3.2.2 The subroutine ‘initfa’

The subroutine ‘initfa’ is located in file ‘base.f’ and has the following declaration:

```
subroutine initfa(block,imax,jmax,kmax)
```

Here *block* is the number of the current block, and $imax, jmax$ and $kmax$ are the number of grid vertices in each direction +1. All parameters are integers.

3.2.3 The subroutine ‘snowin’

The subroutine ‘snowin’ is located in face ‘winbas.f’ and it has the following declaration’:

```
subroutine snowin(block,side,nowin)
```

It *must* be called for each face in each block, and upon call the parameters should contain:

- *block* The number of the current block.
- *side* The number of the current face according to the above definitions.
- *nowin* The number of windows on this face.

All parameters are integers.

3.2.4 The subroutine ‘setwin’

The subroutine ‘setwin’ specifies the dimension of and the boundary types for a window and must be called for each window on each block. The routine can also be used during the computation to, for example, change a boundary type for a window, simulating, for example, the closing of a valve.
The declaration of the routine is the following:

```
subroutine setwin(block,side,window,wmst,wnst,wmend,wnend,
.   wbtyp,wbmst,wbnst,wmdir,wndir,wblock,wside,wshift)
```

and upon call, the parameters should contain the following:

- *block* The number of the current block.
- *side* The number of the current face according to the above definitions.
- *window* The number of the current window.
- *wmst* The lower left m coordinate of the window.
- *wnst* The lower left n coordinate of the window.
- *wmend* the upper right m coordinate of the window.
- *wnend* the upper right n coordinate of the window.

- *wbtyp* the boundary type for the window taken from the boundary library 'action.f'. $wbtyp = 1$ corresponds to a multiblock boundary, $wbtyp = 2$ to a wall etc. See below.

- *wbmst* If the boundary type for the window is not a multiblock boundary this parameter should be set to 1. Otherwise it should be set to the lower left m coordinate for the matching multiblock boundary window in **that window's coordinate system**.

- *wbnst* If the boundary type for the window is not a multiblock boundary this parameter should be set to 1. Otherwise it should be set to the lower left n coordinate for the matching multiblock boundary window in **that window's coordinate system**.

- *wmdir* If the boundary type for the window is not a multiblock boundary this parameter should be set to 1. Otherwise if the multiblock windows are aligned in opposite m directions, the parameter should be set to -1. If the matching windows are aligned in the same m directions, it should be set to 1.

- *wndir* If the boundary type for the window is not a multiblock boundary this parameter should be set to 1. Otherwise if the multiblock windows are aligned in opposite n directions, the parameter should be set to -1. If the matching windows are aligned in the same n directions, it should be set to 1.

- *wblock* The number of the block of the matching window. Should be set to 1 if the boundary type is not multiblock.

- *wside* The number of the face of the matching window. Should be set to 1 if the boundary type is not multiblock.

- *wshift* If the matching windows are aligned, so that the m direction of a window corresponds to the n direction of the matching window, this variable should be set to .true. otherwise .false. The .true. option is probably extremely seldom used and is included only for generality. If *wshift* is set to *.true.*, we have to redefine the meaning of *wmdir* and *wndir*. The variable *wmdir* should be set to -1 if the m index on the *local* window is aligned in the opposite direction of the n index of the *remote* window. If the m index on the local window is aligned in the same direction as the n index on the remote window, *wmdir* should be set to 1. In the same manner, the variable *wndir* should be set to -1 if the n index on the *local* window is aligned in the opposite direction of the m index on the *remote* window and to 1 if the n index on the local window is aligned in the same direction as the m index on the remote window. The letters m and n in the variable names *wmdir* and *wndir* are thus *always referring to the indices of the local window, regardless of the value of wshift.*

3.2.5 Action.f - The boundary type library

A boundary type consists of a set of actions to be taken for different variables at the boundary. The actions implemented are:

- 1 - Homogenous Dirichlet.

- 2 - Dirichlet. The value at the boundary is specified in the *user supplied* routine 'setbou'.

- 3 - Homogenous Neumann.

- 4 - $\partial^2\phi/\partial n^2 = 0$ Used for the pressure at, for example, walls.

- 5 - Multiblock.

- 6 - Inflow/Outflow dependent action. For inflow, the action 2 is imposed, and the boundary value is specified in user supplied subroutine 'setbou'. For outflow, a homogenous Neumann action (3) is imposed. The *user supplied* routine setio determines which cell faces are inflow faces and which cell faces are outflow faces.

It is important to be aware of the difference between an 'action' and a 'boundary type'. A boundary type is built up by specifying which actions should be taken for different variables. This is done by setting different fields in the two-dimensional array *whtodo*. Some standard boundary types are already supplied in 'action.f' These are:

- 1 - General multiblock boundary, action 5 is imposed for all variables.
- 2 - Wall boundary. Action 3 is imposed for a_p, action 1 is imposed for u,v and w, action 4 is imposed for p, action 4 is imposed for pp (sometimes action 3 gives faster convergence than action 4, Beware, however that this may not be a consistent boundary condition), finally action 1 is imposed for k and ε.
- 3 - Symmetry in yz - plane (x constant). Action 3 is imposed for all variables but u, for which a homogenous Dirichlet action (1) is imposed.
- 4 - Symmetry in xz - plane (y constant). Action 3 is imposed for all variables but v, for which a homogenous Dirichlet action (1) is imposed).
- 5 - Symmetry in xy-plane (z constant). Action 3 is imposed for all variables but w, for which a homogenous Dirichlet action (1) is imposed.
- 6 - Given profile boundary. Action 3 is taken for a_p, (which in fact always should be done except for multiblock boundaries). Action 2 is imposed for u, v and w, and the velocity profile is thus given by the user supplied subroutine 'setbou'. Action 4 is imposed for p, action 3 is imposed for pp, and action 6 is imposed for k and ε. The reason action 6 is imposed for the two last variables is, that this boundary type can be used for both given inflow and given outflow profiles, and a homogenous Neumann condition should be imposed on k and ε at outflow boundaries. To be able to use this condition, the user must supply the routines 'setbou' and 'setio'. The purpose of these routines are described below. In fact, in the momentum equation, upwinding is used at the outflow boundary, so that the outflow boundary condition in the momentum equations is $\partial u/\partial$n= 0. This does not affect the accuracy of the solution.
- 7 - Outflow profile boundary. When this boundary type is specified, the condition $\partial u/\partial$n= 0. is imposed for the momentum equations at outflow windows (action 6). This means, that the convection at the outflow window will be adjusted accordingly and a velocity profile will be built at the boundary. For inflow windows, the routine 'setbou' will be called to determine the velocity profile. As usual the routine 'setio' will determine which cell faces are outflow and which are inflow. When the outflow window is small compared to the computational domain, the solution will converge very slowly, because of the fact that the calculated outflow convection may not have the same value as the inflow convection. To avoid this, the variable *adconv* may be set to *.true.* (see below).

Now, to specify a boundary type for a window, the user only sets the parameter *wbtyp* in the call to 'setwin' to the desired boundary type. If a window corresponds to a wall boundary, for example, *wbtyp* should be set to two.
It is straightforward to implement new boundary types given the actions defined above. The implementation of some of the standard boundary types looks like this:

```
subroutine action
```

```
       include 'common.for'

c 1 - Multiblock

      whtodo(1,0)=5
      whtodo(1,u)=5
      whtodo(1,v)=5
      whtodo(1,w)=5
      whtodo(1,p)=5
      whtodo(1,pp)=5
      whtodo(1,te)=5
      whtodo(1,ed)=5

c 2 - Wall

      whtodo(2,0)=3
      whtodo(2,u)=1
      whtodo(2,v)=1
      whtodo(2,w)=1
      whtodo(2,p)=4
      whtodo(2,pp)=3
      whtodo(2,te)=1
      whtodo(2,ed)=1
```

Note that the value 0 is used for a_p, and that, the action for a_p *must* be homogenous Neumann (3) except in the multiblock boundary type. This is because this action is also used when, for example, the volumes of the ghost cells are computed, which is done by linear extrapolation.

3.2.6 The user specified routines 'setbou' and 'setio'

The purpose of the routine 'setbou' is to specify values for certain quantities at the boundaries, for example a velocity profile or a pressure distribution. It's declaration is as follows:

```
       subroutine setbou(block,iside,iwin,nmst,nnst,nmend,nnend,variab,
      .  inner,res)

      include 'common.for'
      integer variab,block
      real res(0:msiz,0:msiz),inner(0:msiz,0:msiz)
```

The routine is called from the program with the following parameters specified:

- *block* The number of the block for which the value at the boundary is to be specified.
- *iside* The face for which the value at the boundary is to be specified.
- *iwin* The window for which the value at the boundary is to be specified.
- *nmst* The lower left m coordinate for that window.
- *nnst* The lower left n coordinate for that window.

- *nmend* The upper right m coordinate for that window.
- *nnend* The upper right n coordinate for that window.
- *variab* The variable for which the boundary value is to be specified.
- *inner* An array (m,n) containing the values of the variable in the cell layer *just inside* the boundary.

The values at the boundary for the particular variable should be placed in res(m,n) with (m,n), as before being the local coordinate system for the window.
Let's take a concrete example: We have specified a domain consisting of two blocks. On the west face of block 1 we have specified an inflow window (boundary type 6), for which we want the velocity to be (1,0,0). The turbulent quantities k and ε are set to zero on the inflow boundary. On the east face of block 2 we have specified an outflow window (boundary type 6) for which we want the velocities to be 1.1 times the velocity in the cell layer in the computational domain just inside the window. The turbulent quantities automatically gets a homogenous Neumann condition on the outflow boundary. There are no other windows in the domain that would give rise to a call to 'setbou'. The routine may then be written in the following way:

```
      subroutine setbou(block,iside,iwin,nmst,nnst,nmend,nnend,variab,
     .  inner,res)

      include 'common.for'
      integer variab,block
      real res(0:msiz,0:msiz),inner(0:msiz,0:msiz)

      if (variab .eq.  u) then

         do mi=nmst+1,nmend
            do ni=nnst+1,nnend
               if (block .eq.  1) then
                  res(mi,ni)=1.
               else
                  res(mi,ni)=1.1*inner(mi,ni)
               end if
            end do
         end do

      else if (variab .eq.  v .or.  variab .eq.  w) then

         do mi=nmst+1,nmend
            do ni=nnst+1,nnend
               if (block .eq.  1) then
                  res(mi,ni)=0.
               else
                  res(mi,ni)=1.1*inner(mi,ni)
               end if
            end do
         end do

      else if (variab .eq.  te .or.  variab .eq.  ed) then

         do mi=nmst+1,nmend
            do ni=nnst+1,nnend
```

```
            res(mi,ni)=0.
         end do
      end do

   end if

   return
   end
```

Since 'setbou' will never be called for other blocks than 1 and 2 or for other variables than u,v,w,k and ε in this particular case, we need not specify what happens with the other variables. Note that we never use the values of *iside* and *iwin*.
The subroutine setio is used to specify which cell faces are inflow and which are outflow. This is so that the internal routines will know when to apply a homogenous Neumann and when to apply a Dirichlet boundary condition when action 6 has been specified for a variable. Since we have chosen standard boundary type 6 for our variables at the inflow and outflow windows, action 6 is automatically imposed for k and ϵ (See file 'action.f'!) We therefore need to specify the subroutine 'setio'. We assume that there is no inflow at all on our outflow window on the west face of block 2. By now specifying inflow on our inflow face and outflow on our outflow face, the routine 'setbou' will be called for the inflow face, setting k and ε to zero, and by specifying outflow on the outflow face, a homogenous Neumann condition is applied.
The subroutine 'setio' should have the following declaration:

```
      subroutine setio(block,iside,iwin,nmst,nnst,nmend,nnend,
     .  res)
      include 'common.for'
      integer res(0:msiz,0:msiz)
      integer block,iside,iwin,nmst,nnst,nmend,nnend
```

The meaning of the parameters are the same as for the routine 'setbou' with the exception that *variab* and *inner* is left out. Upon return, the routine 'setio' should specify 1 for an outflow cell face and -1 for an inflow cell face in the array $res(m,n)$. In our case it would be something like this:

```
      subroutine setio(block,iside,iwin,nmst,nnst,nmend,nnend,
     .  res)
      include 'common.for'
      integer res(0:msiz,0:msiz)
      integer block,iside,iwin,nmst,nnst,nmend,nnend

      do mi=nmst+1,nmend
         do ni=nnst+1,nnend
            if (block .eq.  1) then
c     Inflow cell faces
               res(mi,ni)=-1
            else
c     Outflow cell faces
               res(mi,ni)=1
            end if
         end do
```

```
end do

return
end
```

Again, ‘setio’ will not be called for other windows than our inflow and outflow windows.

3.2.7 Correcting the outflow convection

If boundary types 6 or 7 are used for an outflow window the total outflow convection may be different from the total inflow convection. This will lead to a global continuity error and slow convergence. If the variable *adconv* is set to *.true.* the program will multiply the outflow profiles with a factor C, determined from

$$C = \frac{\sum_{\text{inflow faces}} -\rho\vec{U}\cdot\vec{n}A}{\sum_{\text{outflow faces}} \left|\rho\vec{U}\cdot\vec{n}A\right|} \tag{1}$$

3.2.8 Specifying the pressure reference node

Even if there is a Dirichlet boundary condition for the pressure, one usually has to specify a node in a block for which the pressure is zero. This is done in the same way as in standard CALC, but one also has to specify a block in which the zero-pressure node is located. If there is a homogenous Dirichlet boundary condition for the pressure somewhere in the computational domain, The pressure reference node should be located in the node layer next to this boundary. Let’s assume that we want the pressure to be zero in node (2,3,4) in block 5. This is done the following way:

```
ipref=2
jpref=3
kpref=4
call setref(5)
refnod=.true.
```

If we do have a Dirichlet condition for the pressure somewhere on a boundary, and we do not want to have a pressure reference node, $refnod$ should be set to zero. $.false.$

3.2.9 Optimizing for two-dimensional calculations

If we want to perform a two-dimensional calculation it is possible to greatly reduce the amount of memory needed and also the computational time by setting the flag *twodim* to *.true.* *before* the first call to ‘initfa’. This can however only be done if the computational domain is specified in the x-y plane and in the i-j plane. The effects of setting *twodim* to .true. are the following:

- The function ‘dphidz’ always returns zero.
- The number of ghost cell layers is reduced from two to one on the high an low faces.
- The TDMA coefficients a_h and a_l are never computed, but are always set to zero.

For the high and low faces (5 and 6) the boundary type 5 (x-y symmetry) *must* be specified if *twodim* is set to *.true.*
For all other calculations, *twodim* should be set to $.false.$

3.2.10 Specifying the number of ghost cell layers used

Some discretization schemes, like QUICK use both ghost cell layers. Other schemes like the hybrid scheme, however, need only use one layer. It is possible to save some computational time if schemes that only use one ghost cell layer are used by setting the values in the variable $numlay(1:6)$ to 1. The six elements in the array represent the six different faces, but for all practical cases all elements should be set to the same value. If a scheme that uses both ghost cell layers is used, the elements in *numlay* must be set to two. Thus, for QUICK, Van Leer, etc., include the following

```
do i=1,6
   numlay(i)=2
end do
```

For the hybrid scheme and other schemes only using the nearest neighbor, include the following instead:

```
do i=1,6
   numlay(i)=1
end do
```

It should be mentioned, that with a modest number of blocks, the CPU-time saved by setting *numlay* to 1 is negligible. The safest thing to do is to always have *numlay* set to 2.

3.2.11 Specifying the time discretization scheme.

The time discretization schemes available are the Backward Euler scheme (first order) and a second order BDF scheme. The first timestep is always taken with the Backward Euler scheme. Both schemes have the same stability properties as the continous problem. The second order BDF scheme may, however, cause scalar quantities to become negative and should therefore be used with care when applied to, for example, the turbulent quantities. The scheme is specified in the one-dimensional array *tschem*. If we, for example, want to use the second order BDF scheme for the velocities and the Backward Euler scheme for the turbulent quantities, we use the following assignments (2 for second order BDF and 1 for Backward Euler):

```
tschem(u)=2
tschem(v)=2
tschem(w)=2
tschem(te)=1
tschem(ed)=1
```

3.2.12 Initializing the flowfield

This is done in the user-specified routine 'flowini'. A default routine comes with the code. The values specified here will also be the initial values for a transient computation.

3.2.13 Initializing the turbulent quantities

If the flag *initur* is set to *.true.* in the routine 'setup', a call will be made to the built in routine 'turini' after the flowfield has been specified. This routine will provide guessed values for the turbulent quantities.

3.2.14 Selecting the blocks to use

In some calculations, it may not be nessecary to use all the blocks all the time, for example in transient calculations where one wishes to disconnect some part of the domain at certain timesteps. To make this possible there is a boolean array called *use* in which one specifies which blocks to use and which not to use. Let's assume we want to use block 1 and 3 but not block 2. This would be done as follows:

```
use(1)=.true.
use(2)=.false.
use(3)=.true.
```

The default is to set the whole vector to *.true.*

3.2.15 Setting convergence criterion

As in standard CALC the residual reference value is set in the vector $reref$. Sometimes, however, the residual of a specific variable is significantly larger than the residuals of the other variables, without affecting the mean flowfield. This is, for example sometimes the case for ε. It is possible to select which residuals are significant by setting the corresponding field in the logical vector *mcrit* to *.true.*, which is the default. If, for example, ε is to be left out, the following line should be included in the routine 'setup':

```
mcrit(ed)=.false.
```

3.2.16 Debugging block connectivity

The routine 'conc' performs a printout of the mass flux through all the faces of all the blocks. If the block connectivity is set up correctly, the outflow through a multiblock boundary should be exactly the same as the inflow through the corresponding boundary in the neighboring block (down to machine precision) if, of course, the gridpoints in the different blocks at the multiblock boundary are perfectly aligned. The routine is called automatically at iteration number *icheck*, where the value of *icheck* should be specified in the routine 'setup'.

3.2.17 Specifying constants for Rhie & Chow interpolation

The Rhie & Chow interpolation uses a_p as a weighting constant for the pressure gradients in the computation of the mass flux at the cell faces. This gives, however, different solutions if one uses different timesteps and, in particular, a result from a steady calculation will change if set as initial data in a transient computation. This is because a_p changes with the length of the time step. To remedy this problem, a new variable, *aprc*, is used as the weighting value instead of a_p. The value of *aprc* is, like a_p computed in the routine 'assemb', but instead of using the standard values of the timestep, *dt*, the under-relaxation factor urf and the false timestep $dtfals$, the value of *aprc* is computed with the user defined values *dtmin*, $urfmin$ and $dtfmin$. These values should typically be set to the minimum values of the quantities *dt*, urf and $dtfals$ respectively. This is done as default in the file 'setup.f'. However, should the user decide to change one or more of these values it is possible to do so. Remember, however, that a too large difference between for example *dtmin* and *dt* will cause the iteration process in the pressure correction equation to be unstable.

4 Examples

4.1 Specifying setup data with subroutine calls

In the first example the flow domain consists of two cubic blocks as seen in figure 1. The domain is limited by rigid walls except for an inflow window on the high face (5) on block 1 and an outflow window

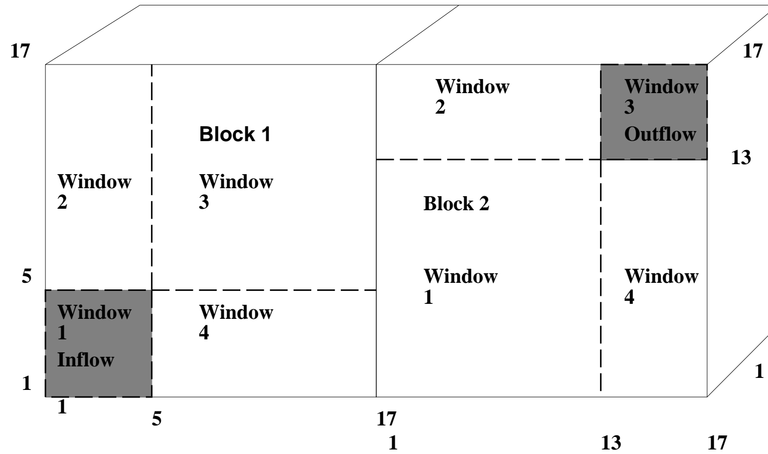

Figure 1:

on the high face (5) on block 2. Since a window has to be rectangular, the high faces on the blocks is here divided into four windows. (It is possible to use only three). The inflow and outflow velocity profiles are fixed to $u = 0$,$v = 0$, and $w = -1$ for inflow and $w = 1$ for outflow: The connectivity and boundary data would then be specified as follows:

```
c       17 grid points in each direction for block 1

        call initfa(1,18,18,18)

c       For block 1, face 1 we have a multiblock window interfacing
c       with face 3 on block 2.  For both blocks, the m index is index j
c       and the n index is index k.  Thus wshift in call to setwin
c       should be set to .false. Increasing index m on block 1 will
c       also increase index m on block 2 (on the interfacing faces.
c       Thus wmdir and wndir should be set to 1.

        call snowin(1,1,1)
        call setwin(1,1,1,1,1,17,17,1,1,1,1,1,2,3,.false.)

c       1 window on face 2-4.  It is a wall.  Lower left corner is (1,1).
c       Upper right corner is (17,17)

        do i=2,4
           call snowin(1,i,1)
           call setwin(1,i,1,1,1,17,17,2,1,1,1,1,1,1,.false.)
        end do

c       For face 5 we have 4 windows

        call snowin(1,5,4)

c       Window 1 is inflow, thus boundary type 6.  it ranges from (1,1)
c       to (5,5)

        call setwin(1,5,1,1,1,5,5,6,1,1,1,1,1,1,.false.)
```

```
      call setwin(1,5,2,1,5,5,17,2,1,1,1,1,1,1,.false.)
      call setwin(1,5,3,5,5,17,17,2,1,1,1,1,1,1,.false.)
      call setwin(1,5,4,5,1,17,5,2,1,1,1,1,1,1,.false.)

      call snowin(1,6,1)
      call setwin(1,6,1,1,1,17,17,2,1,1,1,1,1,1,.false.)


c     17 grid points in each direction for block 2!

      call initfa(2,18,18,18)

c     Faces 1,2 are walls.

      do i=1,2
         call snowin(2,i,1)
         call setwin(2,i,1,1,1,17,17,2,1,1,1,1,1,1,.false.)
      end do

c     Face 3 interfaces with face 1, block3.  Because of symmetry,
c     wshift,wmdir and wndir must be set to .false.,1
c     and 1 respectively.

      call snowin(2,3,1)
      call setwin(2,3,1,1,1,17,17,1,1,1,1,1,1,1,.false.)

      call snowin(2,4,1)
      call setwin(2,4,1,1,1,17,17,2,1,1,1,1,1,1,.false.)

      call snowin(2,5,4)
      call setwin(2,5,1,1,1,13,13,2,1,1,1,1,1,1,.false.)
      call setwin(2,5,2,1,13,13,17,2,1,1,1,1,1,1,.false.)

c     Window 3 is outflow, thus boundary type 6.  it ranges from (13,13)
c     to (17,17)

      call setwin(2,5,3,13,13,17,17,6,1,1,1,1,1,1,.false.)

      call setwin(2,5,4,13,1,17,13,2,1,1,1,1,1,1,.false.)

      call snowin(2,6,1)
      call setwin(2,6,1,1,1,17,17,2,1,1,1,1,1,1,.false.)
```

4.2 Reading setup data from data files

Instead of specifying the computational domain and the block connectivity by calls to subroutines, it is possible to specify the data in two datafiles, one containing the grid layout and one containing the block connectivity and the boundary specifications. The mesh is read in from the file "multimesh" by a call to the routine 'rdmesh', which has no parameters. The file "multimesh" should have one of two formats: For two-dimensional calculations the following format should be used: (Text inside {}'s indicates that

it should be replaced by a number. Don't include the {}'s in the file. Text inside `<>`'s indicates that description follows.)

```
{number of blocks} T
<Block1>
<Block2>
<Block3>
...
```

where a `<block>` should be replaced with

```
{nim1}{njm1}
{x(1,1)}{y(1,1)}
{x(2,1)}{y(2,1)}
{x(3,1)}{y(3,1)}
{x(4,1)}{y(4,1)}
...
{x(1,2)}{y(1,2)}
{x(2,2)}{y(2,2)}
...
```

The other format is for three-dimensional calculations and reads

```
{Number of blocks} F
<block1>
<block2>
<block3>
<block4>
...
```

and a `<block>` is now specified by

```
{nim1}{njm1}{nkm1}
{x(1,1,1)}{y(1,1,1)}{z(1,1,1)}
{x(2,1,1)}{y(2,1,1)}{z(2,1,1)}
{x(3,1,1)}{y(3,1,1)}{z(3,1,1)}
...
{x(1,2,1)}{y(1,2,1)}{z(1,2,1)}
{x(2,2,1)}{y(2,2,1)}{z(2,2,1)}
...
{x(1,1,2)}{y(1,1,2)}{z(1,1,2)}
...
```

Note that what causes the routine 'rdmesh' to select between the two formats is the letter behind the specification of the number of blocks. A 'T' will select the two-dimensional format and a 'F' will select the three-dimensional. This letter is also read into the logical flag 'twodim', so that the memory usage will be optimized for the calculation specified.
The block connectivity is read from the file "boundary" when a call to the routine 'bourd' is made. This must be done *after* a call to 'rdmesh'.
The file "boundary" should have the following format:

```
<block1>
<block2>
```

```
<block3>
...
```

where now a <block> has the following format:

```
<face1>
<face2>
<face3>
<face4>
<face5>
<face6>
```

A <face> is specified as follows:

```
{number of windows} <window1>
<window2>
<window3>
...
```

and, finally, a <window> is specified as follows

```
{wmst}{wnst}{wmend}{wnend}{wbtyp}
<<{wbmst}{wbnst}{wmdir}{wndir}{wblock}{wside}{wshift}>>
```

where the names have exactly the same meaning as the parameters to the routine ‘setwin’. Text inside <<>>’s indicates that this information should only be included if $wbtyp = 1$ that is, if the window is a multiblock window. The information inside the <<>>’s must be excluded otherwise.
One may insert blank lines in the files, but no comments.
As an example we present a “boundary” file for the flowcase specified above. Here we include comments preceded with a percent sign only for readability.

```
%Block 1, face 1 one window
1

%window 1
1 1 17 17 1
1 1 1 1 2 3 F

%Face 2 - 4, one window each.  boundary type is wall, therefore
%multiblock information is excluded
1
1 1 17 17 2

1
1 1 17 17 2

1
1 1 17 17 2
```

```
%Face 5, four windows.  No multiblock window information
4
1 1 5 5 6
1 5 5 17 2
5 5 17 17 2
5 1 17 5 2

%Block 2, Face 1 to 2 are walls, 1 window each
1
1 1 17 17 2

1
1 1 17 17 2

%Face 3, one multiblock window
1
1 1 17 17 1
1 1 1 1 1 1 F

%Face 4, wall, one window
1
1 1 17 17 2

%Face 5, four windows
4
1 1 13 13 2
1 13 13 17 2
13 13 17 17 6
1 17 13 2 1

%Face 6, one window; wall.
1
1 1 17 17 2
```

5 Moving mesh specification

At the beginning of each timestep, the routine 'main' checks if the global flag *mmesh* is set to true. If it is, the user - specified routine 'mvmesh' is called. The routine has the following declaration:

```
subroutine mvmesh(block,time)
```

The parameters *block* and *time* are in - parameters and from these parameters, the routine should specify

- The grid vertex coordinates *xc*,*yc* and *zc* if they differ from the startup setting in 'setup'.
- The control volume face velocities in the global arrays *fue*, *fve*, *fwe*, *fun*, *fvn*, *fwn*, *fuh*, *fvh* and *fwh*. The second letter in the variable names indicates the velocity component and the third letter the face direction.
- The global change rate of density, $\partial\rho/\partial t$, stored in the scalar variable *ddendt*.

The routine may also do other things, such as change in connectivity, boundary conditions etc.
If the flag *mmesh* instead is set to *.false.*, the routine 'main' will call the built - in routine *nomove* which sets the control volume face velocities and the change rate of density to zero.

Part II
Technical description

6 Brief multiblock subroutine description

6.1 Selecting a block

To save storage space, all data is stored in one-dimensional arrays. Since a block is a three-dimensional structure, a mapping routine called 'key' is provided. Imagine that you would like to address the velocity in node i, j, k in block p. This would then be done in the following way:

```
call key(p,lst,ist,dummy,dummy,dummy)
cursor = lst(k)+ist(i)+j
phi(cursor,u)= ...
```

An index-variable to the one-dimensional arrays holding the values is in the following called a 'cursor'. Note that there are three dummy variables to 'key' that are not currently used. The details of the routine will be described later.

6.2 Data types

Before we enter the dark world of multiblock boundary routines, we have to discuss some data-types. One is the *layer buffer* which is a two-dimensional quadratic array of size *msiz* specified in the file 'common.for', and is used to temporarily hold values of a layer of cells in the computational domain, or a layer of cursors pointing to such a cell layer. Since Fortran does not recognize an integer or a floating point value unless explicitly told to do so, we may se these buffers as possible containers of either integers (cursors) or floating point values. There are five such buffers declared in 'common.for': $layer1$, $layer2$, *extra*, $ma1$, $ma2$. The buffers $ma1$ and $ma2$ are primarily used for cursors and the others for floating point values (Boundary information).
Another important data type is the *ghost cell buffer* which is a three dimensional array of size *msiz* x *msiz* x 2. The ghost cell buffer is ment only for floating point values and is used to build up the ghost cell layers in before they are transferred back to the actual block for which they were built up. There is only one ghost cell buffer declared in 'common.for' and it is called $cface$.
Finally we shall describe the boundary data base, which can be seen as an array(1:6,1:2,0:nphit) of *layer buffers*. That is one layer buffer for each face, each of the two layers near a boundary, and each of the variables, including one extra *variable location* for a_p, which is number 0. However, this would demand an excessive use of storage space, and therefore the program keeps track of how much storage space each face on each block requires. The actual declaration of the base type is the following (located in file 'base.f')

```
common /fbase/ base(1:ifsiz,2,0:nphit)
```

and the parameter $ifsiz$ should be adjusted so that there is room to store all faces in the computational domain. More about this later. What is important to know is that it is possible to store and retrieve a layer buffer in the boundary data base by specifying

- The layer buffer,
- Which block it belongs to,

- Which face it represents,
- Which layer it is containing (1 or 2) and finally
- Which variable it is containing (0..) where 0 is ment for a_p.

All routines for managing the boundary data base is located in the file 'base.f'.
It should be mentioned here, that it is not really nessecary to have a boundary data base, since there is no need to temporarily store certain cell layers; they may be collected directly from the computational field. However, in a typical three-dimensional application, the extra CPU time overhead for managing such a data base is very small (about 1%) and it is primarily designed to simplify the extension of the code to a multi-workstation system where one workstation holds the boundary data base, and the others are solving one block each.

6.3 The routine 'store' - store variable layer

The subroutine 'store' stores the contents of the desired cell layer into a layer buffer. It's declaration is the following:

```
subroutine store(side,variab,curmat,layer,imax,jmax,lmax,any,ifany)
```

Upon call, the following conditions should be met:

- The routine 'key' should have been called to specify the current block.
- The parameter *side* should contain the face for which the variables should be stored(1-6).
- The parameter 'variab' should contain the number of the variable to be stored, and 0 if a special quantity should be stored (for example a_p, *convh* or whatever desired).
- The parameter *curmat* is the name of the layer buffer.
- The parameter *layer* is the number of the layer (1 is closest to the face and 2 is the next one in the computational domain. It is in fact possible to store a ghost cell layer by specifying 0 for the inner ghost cell layer and -1 for the outer.
- The parameters *imax*, *jmax*, *lmax* controls the number of values stored in a particular direction. These are usually set to $nim1(kblock)$, $njm1(kblock)$ and $nkm1(kblock)$. Depending on which face is specified, only two of these parameters are used. The routine automatically keeps track of which ones to use.
- The parameters *any* and $ifany$ are used to store another variable than the ones stored in $phi(..,..)$. If another variable is to be stored into the layer buffer, *any* should hold the name of the variable, for example a_p and $ifany$ should then be set to *.true*. If one of the variables in $phi(..,..)$ is to be stored, $ifany$ should be set to $.false.$ and the parameter *variab* should, as described above contain the number of the variable.

Finally it should be noted that 'store' transfers a cell layer into a *layer buffer*. Not to the boundary data base.

6.4 The routine 'sind' - store index layer

The subroutine 'sind' stores the cursors of all the cells in a layer into a layer buffer. It's declaration is the following:

```
subroutine sind(side,curmat,layer,imax,jmax,lmax)
```

The parameters are the same as to the routine 'store' except that the variable references are excluded. Note that the information put in the layer buffer *curmat* now should be treated as integers (cursors) rather than floating point values. An example of the flexibility of this routine is the following short code, which computes the total inflow from the computational domain to the cell layer just below the high face:

```
call sind(5,ma3,1,nim1(kblock),njm1(kblock),nkm1(kblock))
inflow=0.
do n=2,njm1(kblock)
   do m=2,nim1(kblock)
      io=ma3(m,n)
      inflow=inflow+smp(io)-convh(io)
   end do
end do
```

6.5 The routine 'facret' - Transfer ghost cell buffer back to block

The routine 'facret' is used to transfer *one* of the layers in the ghost cell buffer to the ghost cells of a block. Unfortunately, for this routine the ghost cell layers has been numbered the other way around than for the 'store' routine. The ghost cell layer closest to the computational domain is number 2 and the one most far out is number 1. The irritated user is hereby granted to modify the routine and all it's callers to use the same convention as does the routine 'store'.
The routine declaration has the following appearance:

```
      subroutine facret(curfac,side,variab,any,ifany,imax,jmax,
     .   lmax,layer)
```

Upon call, the following conditions should be met:

- The routine 'key' should have been called to specify the current block.
- The parameter $curfac$ is the ghost cell buffer.
- The parameters *side*, *variab*, *any*, $ifany$, *imax*, *jmax* and *lmax* are defined in the same way as for the routine 'store'.
- The parameter 'layer' should contain the number of the ghost cell layer to be transferred to the block (1 or 2) as described above.

6.6 The ghost cell value extrapolation routines

There are a number of routines used to build up the ghost cell values in the ghost cell buffers, and all of them are located in the file 'wallset.f' The routines are used by the general ghost cell layer building routine 'bound' and they would probably never be used directly by the user. For completeness a description is included here. The routines are:

6.6.1 The routine 'tface' - Build ghost cell values from a remote face layer buffer

This routine is used to set the ghost cell values in a ghost cell buffer from a layer buffer with values from another face, i. e. it is for multiblock boundaries. It's declaration is:

```
      subroutine tface(curfac,la1,la2,mst,mend,mdir,
     .   nst,nend,ndir,bmst,bnst,lshift,nolay)
```

Upon call the following conditions should be met:

- The parameter *curfac* should contain the ghost cell buffer to which the values are transferred.
- The parameter $la1$ should contain the layer buffer that contains the layer of values closest to the face on the remote face.
- The parameter $la2$ should contain the layer buffer that contains the other layer of values from the remote face.
- The parameter *mst* should contain the first m index of the *ghost cell buffer* for which values should be transferred minus one.
- The parameter *mend* should contain the last m index of the *ghost cell buffer* for which values should be transferred.
- The parameter *mdir* should be set to 1 if the m directions of the remote face and the face for which the ghost cell layer is built have the same n directions. It should be set to -1 if the m directions are opposite.
- The parameters *nst*, *nend*, *ndir* are the same as *mst*, *mend*, *mdir* but for the n index.
- The parameters *bmst*, *bnst* should be set to the lowest m and n indices respectively that should be transferred from the remote face, The data layers of which are stored in $la1$ and $la2$. They should be specified in the *remote face's* coordinate system.
- The parameter *lshift* is a logical parameter and should be set to *.true.* if the m index for the face for which the ghost layer buffer is built corresponds to the n index of the remote face. It should be set to *.false.* otherwise.
- The parameter *nolay* specifies the number of layers to transfer. If *nolay* is set to 1, the parameter $la2$ may be replaced with a dummy parameter, since it is never used.

As an example we imagine a situation where we shall transfer values from face 3 of block 2 to build a ghost cell layer on face 1 of block 1. The m and n indices coincide, but when the n index is incremented on face 1 of block 1, the n index on face 3 of block 2 should be decreased to match the other block.
The transferring code should look something like this if the variable to be transferred is, for example the pressure correction.

```
c       Select block 2.

        call key(2,lst,ist,nim1(2),njm1(2),nkm1(2))

c       Store the values of pp at face 3 of block 2 in the layer buffer
c       layer1 (closest to the face.)  Store the next layer in layer2

        call store(3,pp,layer1,1,nim1(2),njm1(2),
       .   nkm1(2),dummy,.false.)
        call store(3,pp,layer2,2,nim1(2),njm1(2),
       .   nkm1(2),dummy,.false.)

c       Call tface to transfer the values to the ghost cell buffer
c       'cface'.  The m direction corresponds to the i index and the
c       n direction corresponds to the k index, which explains the
c       use of 'njm1' and 'nkm1'.  ndir and mdir are set to -1.  Two
c       layers are transferred.  Note that we use njm1 and nkm1 for the
c       block for which the ghost cell layer is built!
        call tface(cface,layer1,layer2,1,njm1(1),-1,
```

```
     .   1,nkm1(1),-1,1,1,.false.,2)

c       Transfer the ghost cell buffer 'cface' to the ghost layers of
c       block 1, face 1

        call key(1,lst,ist,nim1(1),njm1(1),nkm1(1))
        call facret(cface,1,pp,dummy,.false.,nim1(1),
     .   njm1(1),nkm1(1),1)
        call facret(cface,1,pp,dummy,.false.,nim1(1),
     .   njm1(1),nkm1(1),2)
```

6.6.2 The routine 'xtrapl' - Transfer values from a layer buffer to a ghost cell buffer

The routine 'xtrapl' is used to do a simple transfer of values from a layer buffer to a ghost cell buffer, and is typically used for extrapolation of values at boundaries corresponding to, for example, a symmetry boundary.
The routine has the following declaration:

```
        subroutine xtrapl(curface,myface,mst,mend,nst,nend,
     .   nolay)
```

The following conditions should be met upon call:

- The parameter $curface$ is the ghost cell buffer.
- The parameter $myface$ is the layer buffer from which the values are transferred to the ghost cell buffer.
- The parameters *mst*,*mend*,*nst*,*nend* and *nolay* all have the same meaning as in the routine 'tface'.

6.6.3 The routine 'setval'

The routine 'setval' builds ghost cell values, so that by using linear interpolation, one gets a predefined value at the boundary.
The routine has the following declaration:

```
        subroutine setval(curface,myface,vts,mst,mend,nst,nend,nolay)
```

The following conditions should be met upon call:

- The parameters $curface$,*mst*,*mend*,*nst*,*nend*,*nolay* are specified in the same way as for the routine 'tface'.
- The layer buffer $myface$ should contain the layer of values closest to the boundary in the computational domain.
- The layer buffer *vts* - value to set, should contain the desired cell face value for each cell at the face.

6.6.4 The routine 'ioval'

The routine 'ioval' is a combination of 'setval' and of 'xtrapl'. It checks if the cell face corresponding to a cell in the ghost cell buffer is inflow or outflow. If it is inflow, the same thing is done for the ghost cell as would be done in 'setval'. If it is outflow, the same thing is done as would be done in 'xtrapl' The routine thus requires information of which cell face is outflow and which is inflow.
The routine has the following declaration.

```
      subroutine ioval(curface,myface,vts,con,mst,mend,nst,nend,
     .   nolay)
```

As can be seen, the only thing that differs from the routine 'setval' is the extra parameter *con*. This parameter is a layer buffer containing integers that tells the routine which cell faces are outflow and which are inflow. The value 1 denotes outflow and the value −1 denotes inflow. The meaning of the other parameters are described above.

6.6.5 The routine 'xxtra'

The routine 'xxtra' is used to build ghost cell layers so that the normal second derivative of the boundary is zero. The routine requires information from the two layers of cells in the computational domain that are nearest to the block face. The declaration of the routine is the following:

```
      subroutine xxtra(curface,myface,vts,mst,mend,nst,nend,
     .   nolay)
```

Upon call, the layer buffer *vts* should hold the layer closest to the block face and the layer buffer *myface* should hold the other layer. The other parameters follow the same convention as above.
It should be noted that this routine assumes that all the cells concerned have the same size. This is true for the two layers of ghost cells and the cell layer closest to the face in the computational domain, since the ghost cells are given the same dimensions as the cells in this layer, unless it is a multiblock face for which this routine would never be used. The next layer in the computational domain may well have another geometry, and therefore it is appropriate to adjust the values in the layer buffer 'myface' for the geometry before call, so that the right interpolation is made. (See the routine 'bound'!)

6.7 The window information database 'winbas.f'

This subroutine library contains a number of subroutines that are used to store and retrieve information about windows. The reader that is not familiar with the window concept is recommended to first read the *Users guide*. The routines are the following:

```
      subroutine setwin(block,side,window,wmst,wnst,wmend,wnend,wbtyp,
     .   wbmst,wbnst,wmdir,wndir,wblock,wside,wshift)
```

The parameters of this routine has been explained in the users guide. The routine simply stores these values into the window data base.

```
      subroutine getwin(block,side,window,wmst,wnst,wmend,wnend,wbtyp,
     .   wbmst,wbnst,wmdir,wndir,wblock,wside,wshift)
```

This routine retrieves the window data stored in the window data base. The parameters *block*, *side* and *window* should be given upon call. The other parameters are out-parameters provided by the subroutine.

```
      subroutine snowin(block,side,nowin)

      integer function gnowin(block,side)
```

These routines are used to set and get the number of windows on a particular face on a particular block. The routine 'snowin' is typically used by the user as described in the *users guide*. The routine 'gnowin' is used internally in the program to retrieve the information the user has specified.

```
      logical function chkbou(block,side,type)
```

This routine is used internally to check whether *any* of the windows on a particular face on a block has the *boundary type* 'type' (See the *User's guide*). If so, it returns *.true*. Otherwise it returns *.false*.

6.8 The layer buffer data base - 'base.f'

This subroutine library contains the complete multiblock management concerning mapping to one-dimensional arrays, storage of global data etc. The routines included are the following:

```
subroutine sface(curfac,block,side,variab,layer)
```

This routine stores a layer buffer into the boundary data base. Upon call the following conditions should be met:

- The parameter $curfac$ is the layer buffer to be stored.
- The parameter *block* is the number of the block from which the layer in $curfac$ was retrieved.
- The parameter *side* is the number of the face from which the layer was retrieved.
- The parameter *variab* is the variable number (0 for a_p).
- The parameter *layer* is the number of the layer where 1 is the layer closest to the face, and 2 is the next layer.

```
subroutine rface(curfac,block,side,variab,layer)
```

This routine is used to retrieve a layer buffer from the layer buffer data base. The description of the parameters is the same as for the routine 'sface', with the exception that 'curfac' is now an out parameter and is not given by the caller.

```
subroutine cbase
```

This routine is used to clear all information in the boundary data base. All entries are set to zero.

```
subroutine initfa(block,imax,jmax,lmax)
```

This routine should be called by the user when the block connectivity is specified. The description of the parameters may be found in the *User's guide*. The routine initializes all the information that is needed by the routines 'key' and 'fackey' to map the three-dimensional structure of a block to a one-dimensional array. Since the exact amount of memory needed is computed here, the routine is able to warn if the allocated space is too small, i. e. if a recompilation is necessary. If this is the case, the routine stops and warns. If the global flag *twodim* is set to *.true.* when this routine is called, the storage space is managed in a slightly different way. The routine then assumes that only one ghost cell layer is necessary in the k direction, and thus saves a large amount of memory space in the case of a two-dimensional calculation.

```
subroutine key(block,lsta,ista,dummy1,dummy2,dummy3)
```

This routine maps the three-dimensional block into a one-dimensional array. The block is assumed divided into $k-slabs$ containing all the values for a particular k - index. The cursor pointing to the first position in slab k is given in $lsta(k)$. Thus, for block 1, $lsta(0)$, which corresponds to the start of the first ghost cell layer on face 6 (The low side, corresponding to k small) should be equal to zero. The value of $lsta(1)$ would be $0+(nim1(1)+4)\cdot(njm1(1)+4)$ etc. (The extra four corresponds to the ghost cell layers). The value of $lsta(0)$ for block p would be $lsta(0)$ for block $p-1+(nkm1(p-1)+4)\cdot(nim1(p-1)+4)\cdot(njm1(p-1)+4)$. The $k-slabs$ are then divided into $i-rows$. The offset to row i from the start of slab k is stored in ista(i). The cursor pointing to the start of row i in slab k is thus given by lsta(k)+ista(i). In each i -row, the $j-elements$ is stored in order. To conclude all this: If the following call has been made to key:

```
call key(kblock,lst,ist,dummy,dummy,dummy)
```

The cursor pointing to element i, j, k for block *kblock* is given by $lst(k) + ist(i) + j$
An earlier implementation of the routine required three extra parameters. With the implementation of the boundary data base, these parameters became unnessecary and were, for compatibility with earlier routines that used the routine 'key' replaced with dummy parameters. The interested user may remove these dummy parameters from the routine and all its callers.

```
subroutine fackey(block,side,bsstar,mst)
```

This routine is used internally by the routines 'sface' and 'rface'. It maps the two-dimensional array structure of a layer buffer into the one-dimensional array structure used in the boundary layer data base. The block number should be specified in *block* and the face number should be specified in *side*. Upon return, the routine gives the start position in the array for this layer buffer in *bsstar*. The layer buffer is then assumed divided into $m - rows$. The offset to the start of row m from the start-position *bsstar* is stored in $mst(m)$. As an example: Imagine that you would like to get a cursor to where the w-velocity in layer 1 on the east face of block 2, m-index 3 and $n - index$ 4, is stored. Since the east face has number one, we would first make the following call:

```
call fackey(2,1,bsstar,mst)
cursor = bsstar+mst(3)+4
```

We would then address the boundary base array:

```
theval = base(cursor,1,w)
```

Note that the boundary base has two extra indices, one for the layer and one for the variable. These indices are not included in the one-dimensional storage structure because they have fixed length. There are always two indices and $nphi + 1$ variables. It should be mentioned that the end user should *not* use this routine directly. The description is only included to provide a base for the understanding of the implementation.

```
subroutine nmmax(block,side,mmax,nmax)
```

Given the block number and the face number in *block* and *side* this routine returns the number of cells in the *computational domain* in the m and n direction of the face. This is used by routines that operates on the whole face rather than on a window. Note that the indices in the routine name has changed place. This may be confusing.

```
subroutine sendpp(block,ppr)
subroutine setref(block)
integer function getref()
real function getpp()
```

These three routines are provided to make it easier to extend the code to a multicomputer system. With the routine 'setref' The block in which the pressure reference node is located is specified. When the pressure correction has been computed for each block, the reference value of the specified cell (but possibly the wrong block) is sent to the database by the routine 'sendpp'. If the block number matches the one specified with 'setref', the reference value is stored. Otherwise it is discarded. The latest reference value stored may then be retrieved with the function 'getpp'. The number of the block holding the pressure reference node may be retrieved with 'getref'.

```
subroutine zerres(block,ppr)
subroutine setres(block,variab,res)
real function getres(block,variab)
```

These routines take care of the residual storage for each block and variable. The subroutine 'zerres' zeroes all the residuals, the routine 'setres' sets a residual for a given block and a given variable. Finally the function 'getres' retrieves a residual for a given block and a given variable.

6.9 The general ghost cell building routines

The description of the previous routines now provide a foundation for the understanding of the general ghost cell building routines 'bound' and 'sbound'. The routine 'sbound' has the following specification:

```
subroutine sbound(block,variab,any,nlay)
```

This routine scans all the faces of the block 'block' too see if any window on the face has a multiblock boundary type. If so, it stores the required number of layers (1 or 2) in a layer buffer and finally stores the layer buffer in the boundary data base. With the parameter 'variab' the number of the variable in question is specified. If the number is 0, the routine does not take the values from *phi*, but rather from the variable specified in 'any' (usually a_p). The parameter *nlay* is an array(1:6) specifying the number of layers to be stored on each face (1 to 6). There are two arrays of this type specified in the initialization of the program. One is $lay1$ which only contains $1's$ and the other is $lay2$ that only contains $2's$. A typical call to this routine, storing two layers of cell volumes in the boundary data base in the variable location for a_p (number 0) would look like this:

```
call sbound(block,0,vol,lay2)
```

The other routine in this pair is 'bound', and it has the following specification:

```
subroutine bound(block,variab,any,nlay)
```

This routine is used to build a ghost cell layer in a ghost cell buffer and then transfer it to the particular block for which it was built. It loops over all the faces of the block and all the windows of the face. It retrieves information about the window and depending on the window's boundary type collects layer buffers either from the block itself or from the boundary layer database in case of a multiblock boundary type. It then uses the ghost cell value extrapolation routines described above to build a ghost cell buffer, and when all windows on a face are visited, the routine transfers the ghost cell buffer back to the block for which is was built. The parameters are the same as for the routine 'sbound'.
There is also modified version of the routine 'bound' called 'ppboun'. This routine is designed for the pressure correction. It only sets ghost cell values for multiblock windows and is thus somewhat faster. It is used only for the pressure correction and is called from the routine 'relax'.

7 The geometry of the ghost cells

7.1 Required geometrical quantities

Since no variables are solved for in the ghost cell control volumes, the geometrical quantities that need to be present are very few. The following is needed.

- The ghost cell volume.
- The interpolation factors f_x, f_y, and f_z, which are needed for the extrapolation of the pressure and the pressure correction and also for computing the cell center and cell face distances when the QUICK scheme is used.
- The quantities $xksi$, $yksi$, $zksi$, $xeta$, $yeta$, $zeta$, $xzeta$, $yzeta$ and $zzeta$ which are needed for the Rhie & Chow interpolation at multiblock boundaries.

7.1.1 Treatment at multiblock boundaries

At multiblock boundaries the ghost cells are given the same dimensions as the corresponding cell in the neighboring block.

- The ghost cell volume is obtained by storing all volumes for the cells nearest to the multiblock boundary by calling 'sboun' for the volume and put it in the a_p location. When the volumes for all blocks have been stored in the boundary data base, the routine 'bound' is called, giving the ghost cells the right volumes, since a_p is also taken from the neighboring block at multiblock boundaries. This is done directly in the 'main.f' routine.
- The interpolation factors are calculated by first storing the cell center values in the multiblock data base (the routine 'scent') and then retrieving them with the routine 'gcent'. In this way, the ghost cells obtain the same cell center coordinates as the corresponding cell in the neighboring block.
- The computation of the Rhie & Chow interpolation factors across multiblock boundaries are very complicated and bugs may well have entered here. First of all we have nine quantities to exchange, but we cannot expect to have more than six variable locations in the boundary data base. On the other hand the boundary data base has room for two layers of values whereas the Rhie & Chow geometrical quantities are only needed in the inner ghost cells. We therefore use a routine 'jacobi', which, given a number to the geometrical quantity, maps it to a specific variable location and a specific layer in the data base. The other complicated thing is, that dependent on how the blocks are aligned, the geometrical quantities change sign across a multiblock boundary. This is solved using special sign- and direction tables. A rule that should be remembered is that if the signs are not computed correctly by the routine, the mass flux through the boundary is not exactly the same (down to machine precision) for the both blocks, thus indicating a bug in the routine. The opposite is not true however; a difference in mass flux through a boundary when computed in the two neighboring blocks is probably due to incorrect boundary specifications.

7.1.2 Treatment at domain boundaries

At the domain boundaries the ghost cells are given the same dimensions as their nearest nieghbour in the computational domain. This is done in the following way:

- The ghost cell volume is calculated by simply giving it the same value as the nearest neighbor in the computational domain. This is done by calling the routine 'bound' with the volume put in the variable location reserved for a_p, which implies that the volume will be extrapolated to the ghost cells in the same manner as a_p will be when the computations have started.
- The interpolation factors are computed in a rather special way. Instead of setting them to 0.5, which is the value they should have at domain boundaries, the cell center coordinates for the computational domain is computed and also the coordinates for the face centers in the standard CALC way. The coordinates are then 'mirrored' to the ghost cells using the routine 'gcent' which sets a Dirichlet boundary condition for the cell center coordinates with the cell face coordinates as the boundary value, and then using the standard dirichlet boundary extrapolation routine 'setval' to set the ghost cell center coordinates. The interpolation factors are then calculated in a standard manner in the routine 'cweigh' located in 'init.f'.
- The geometrical quantities required for the Rhie & Chow interpolation routines are simply extrapolated in the routine 'ggeom', using the special mapping function 'jacobi' described above. No sign change is required at the domain boundaries.

8 Modifications to standard routines

8.1 The routine 'wallf' - Set wall functions

The routine wallfunc calculates the wall functions for k, ε or the velocities by reading window information from the window data base and calling the routine 'wall' with the correct parameters. The calls to 'wallf' are made from routines in the file 'modify.f' *and* from the routine 'relax', so that the correct viscosity is used in the u discretization matrix. The function itself is located in file 'wf.f'. Note that 'wallfunc' will only call the routine 'wall' if the boundary type *wall (#1)* has been specified for a window on a face, and then only for that window. The declaration of the routine 'wallf' is:

```
subroutine wallf(block,variab)
```

Before a call to the routine, the current block has to be specified with a call to 'key'.

8.2 The routine 'mdcon' - compute *conv* at domain boundaries

The routine 'conv' has been modified to compute the mass flux even on the boundaries of each block, using Rhie & Chow interpolation. However, on the domain boundaries the convection should be computed without Rhie & Chow interpolation. This routine recomputes the convection on the domain boundaries without Rhie & Chow interpolation. It is called from 'modify.f' ('modcon').
It's declaration is

```
subroutine mdcon(block)
```

8.3 The global convection correction routines 'conset' and 'upconv'

The routine 'conset' computes the *global* inflow to and outflow from the computational domain. The routine gets information about which cell faces are inflow faces from the user specified routine 'setio'. Inflow may be both positive and negative, whereas all mass flux through outflow cell faces is considered outflow, although it may flow inwards. This helps stabilizing the outflow correction on $\partial U/\partial n = 0$ boundaries. The routine declaration is:

```
subroutine conset(block,conin,conout)
```

Upon call the current block should be specified in *block*. The routine will then add inflow to the domain to *conin* and add outflow to *conout*. Note that the two latter parameters need not be zero when the routine is called. The routine simply adds the mass flux to the previous value of the parameters. This simplifies multiple calls to the routine in a multiblock environment. The routine is called from 'relax'.
The routine 'upconv' multiplies the convection on outflow faces with a factor. In the implementation in 'relax' The routine 'conset' is first called for each block determining the global inflow and outflow. Then a multiplication factor is computed, so that the outflow will equal the inflow. Finally 'upconv' is called to correct the outflow so that global continuity is achieved. It should, however, be mentioned that in heavily accelerating flows, the Rhie & Chow interpolation will make the flow somewhat compressible. Therefore specifying global continuity may cause the residual of the continuity equation to hang at a value dependent of the acceleration.
The routine declaration is :

```
subroutine upconv(block,mulfac,totout)
```

Upon call, $mulfac$ should contain the multiplication factor (real number), the routine will then add the corrected outflow to the value of *totout*.

8.4 'Upcoef' - Zero domain boundary coefficients

The purpose of this routine is to zero all coefficients a_e, a_n etc. on the domain boundaries, so that a homogenous Neumann condition is fulfilled. The routine is used by the pressure correction routine 'calcp'.

8.5 'Ucoef' - Zero outflow coefficients

This routine zeroes all coefficients on outflow cell faces, so that a homogenous Neumann condition is fulfilled. The routine gets the information about which cell faces are outflow from the user-supplied routine 'setio'. The routine is called from 'modify.f' and is located in file 'Upcoef.f'

8.6 Splitting of the routine 'calcp'

The SIMPLE algorithm usually requires several sweeps with the Gauss - Seidel solver for the pressure correction equation. Experience has shown that the boundary conditions for each block has to be updated after every sweep. This demands a splitting of the routine 'calcp' into four routines.

- 'calcp' sets up the coefficient matrix.
- 'calcp2' performs a solver sweep.
- 'calcp3' updates pressure and conv.
- 'correc' updates the nodal velocities.

8.7 Modifications to 'Coeff'

The QUICK scheme in the 'coeff' routine has been modified to take use the interpolation factors f_x, f_y and f_z to compute the coefficients instead of the cell volume divided by a characteristic area scale. This is because the interpolation factors are available in the ghost cells whereas the area scale is not. The derivation of the expression for the coefficients are given in section 10.1.

8.8 Modifications to 'Update'

In order to accurately solve transient problems a second order multistep time discretization has been implemented. Since the scheme uses the values from two previous timesteps, the first timestep must be taken with the first order BDF scheme (Backward Euler). The second order BDF scheme is obtained by fitting a second order polynomial to the current value and the values from the previous two timesteps, with the boundary condition that the differential equation shall be fulfilled at the current timestep. This yields an implicit method that is A - stable, i. e. it has the same stability properties as the continous Navier - Stokes equations, which implies that arbitrarily large timesteps may be taken, as long as no constraints are placed on the values at the different timesteps. The turbulence equations however have the restriction that the turbulent quantities may not become negative. Due to their stiff behaviour, this scheme is not suitable for these equations. Although it has the desired stability properties, it cannot guarantee that the turbulent quantities may not become negative.

9 Moving mesh routines

The routines 'mvmesh' and 'nomove' has already been described in the *User's guide*, In addition, some other features have been implemented that allows the computations on moving meshes.

- When the dimensions of the geometrical quantities have changed, that is if the flag $mmesh$ is set to true, a call is made to 'init' to recompute the geometrical quantities. The routine 'init', however, uses a lot of variables as temporary storage space, and these variables are needed as a solution

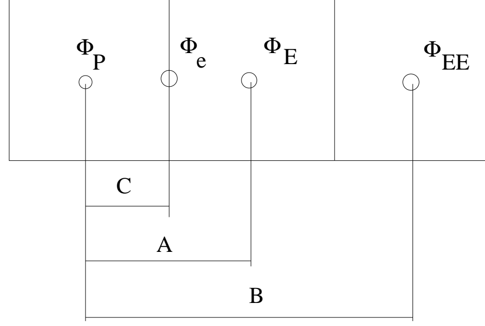

Figure 2: Nomenclature for QUICK discretization scheme.

approximation on the next timestep. Therefore the routine 'sfield' is called before the call to 'init'. This routine is located in file 'misc.f' and saves the flowfield on a temporary file called "fieldsave". After the routine 'init' has made it's computations, the field is read back by the routine 'rfield' located in the same file.

- After a call to 'mmesh', the routine 'concom', located in file 'misc.f' is called. This routine computes the rate of change in volume for each cell and stores the result in the global array *dvoldt*. It also computes the global virtual inflow of mass due to the contraction of the computational domain, (which will of course, due to incompressibility, cause an equal amount of outflow, unless the change rate of density, *ddendt* is different from zero. The global virtual inflow, with the change rate of density taken into account, is then stored in the global variable *consou*. This value is used when the corrected outflow is computed for free outflow boundaries. Note that the value of *consou* may well be negative if the computational domain is expanding.
- The routine 'conv' has been changed to take into account the velocities of the control volume faces when the mass flux is evaluated.
- The calculation of the mass error in the routines 'conv' and 'calcp3' also take into account the rate of change in volume for each cell and also the global rate of change in density.

10 Theoretical explanation of some implementations

10.1 The QUICK scheme

Since QUICK is derived from a second order polynom fitted to the central point, Φ_p , the upwind point, Φ_e and the far upwind point Φ_{ee} we can write

$$\left\{ \begin{array}{rcl} \Phi_P + a_1 A + a_2 A^2 & = & \Phi_E \\ \Phi_P + a_1 B + a_2 B^2 & = & \Phi_{EE} \end{array} \right. \tag{2}$$

$$\left\{ \begin{array}{rcl} a_1 A + a_2 A^2 & = & \Phi_E - \Phi_P \\ a_1 B + a_2 B^2 & = & \Phi_{EE} - \Phi_P \end{array} \right. \tag{3}$$

Now use gaussian elimination to invert the matrix and obtain explicit expressions for a_1 and a_2.

$$a_2 B^2 - \frac{a_2 A^2 B}{A} = \Phi_{EE} - \Phi_P - \frac{B}{A}\Phi_E + \frac{B}{A}\Phi_P \tag{4}$$

$$a_2\left(B^2 - AB\right) = \Phi_{EE} - \Phi_P\left(\frac{B}{A} - 1\right) - \Phi_E\frac{B}{A} \tag{5}$$

$$a_2 = \frac{\Phi_{EE}}{B^2 - AB} + \frac{\Phi_P\left(\frac{B}{A} - 1\right)}{B^2 - AB} - \frac{\Phi_E\frac{B}{A}}{B^2 - AB} \tag{6}$$

$$a_1 = \frac{\Phi_E}{A} - \frac{\Phi_P}{A} - a_2 A \tag{7}$$

$$\tag{8}$$

We now use the following to obtain the QUICK coefficients c_P, c_E and c_{EE}:

$$\begin{cases} \Phi_e = \Phi_P + a_1 C + a_2 C^2 \\ \Phi_e = c_P\Phi_P + c_E\Phi_E + c_{EE}\Phi_{EE} \\ c_P + c_E + c_{EE} = 1 \end{cases} \tag{9}$$

Now identify the coefficients:

$$\begin{aligned} c_E &= \frac{C}{A} - a_2 AC = \\ &= \frac{C}{A} + \frac{AC\frac{B}{A}}{B^2 - AB} + \frac{C^2\frac{B}{A}}{B^2 - AB} = \\ &= \frac{C}{A} - \frac{B}{A}\frac{C^2 - AC}{B^2 - AB} \end{aligned} \tag{10}$$

$$c_{EE} = \frac{AC}{B^2 - AB} + \frac{C^2}{B^2 - AB} = \frac{C^2 - AC}{B^2 - AB} \tag{11}$$

$$c_P = 1 - c_E - c_{EE} \tag{12}$$

Now, we want to adimensionalize the expressions and make the coefficients c_P, c_E and c_{EE} functions of the dimensionless quantities B/A and C/A which are easily computed from the interpolation factors f_x, f_y and f_z. We divide the equation (11) with A^2 and obtain:

$$c_{EE} = \frac{\left(\frac{C}{A}\right)^2 - \frac{C}{A}}{\left(\frac{B}{A}\right)^2 - \frac{B}{A}} \tag{13}$$

$$c_E = \frac{C}{A} - \frac{B}{A}c_{EE} \tag{14}$$

$$c_P = 1 - c_E - c_{EE} \tag{15}$$

and for the east face we easily identify:

$$\frac{C}{A} = f_x(P) \tag{16}$$

$$\frac{B}{A} = \frac{1 - \frac{C}{A}}{f_x(E)} + 1 \tag{17}$$

We observe that by setting $f_x(P)$ and $f_x(E)$ to 0.5, we get the QUICK coefficients for uniform cartesian meshes: $c_E = 0.75$, $c_{EE} = -0.125$ and $c_P = 0.375$.

10.2 Adaptive under-relaxation

The general discretized equation for a control volume may be written

$$a_p\Phi_p - \sum_n a_n\Phi_n = S_u \tag{18}$$

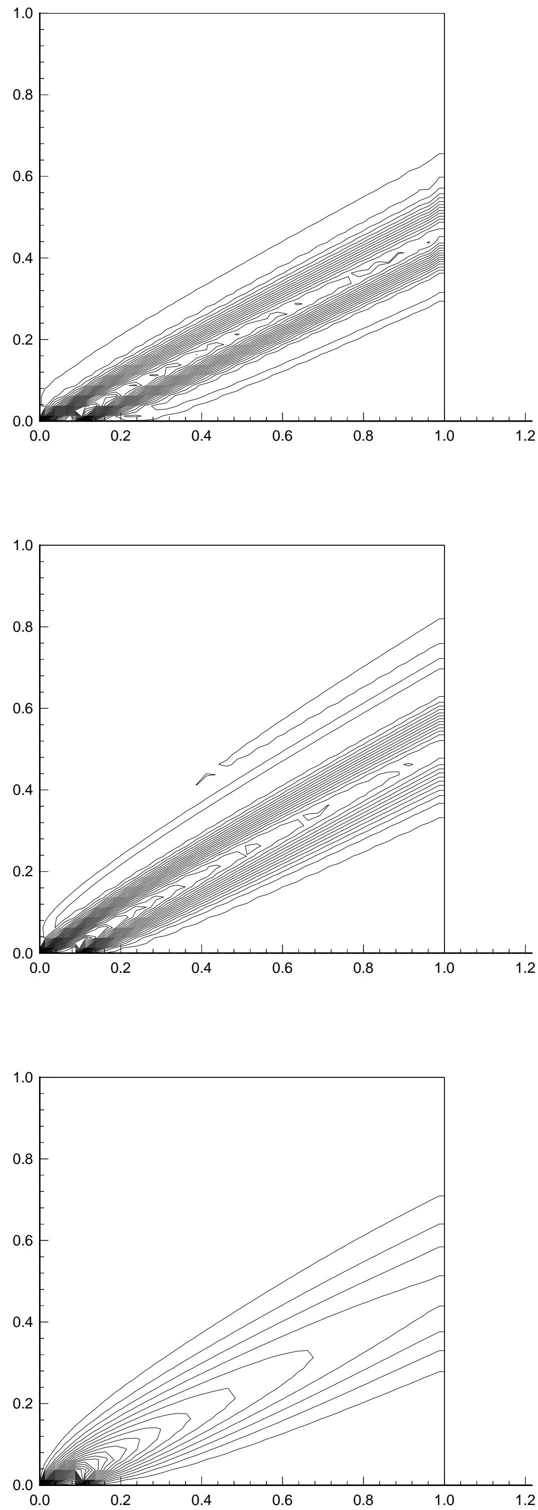

Figure 3: Comparison between QUICK (top), Second order upwind (mid) and HYBRID (bottom) scheme. Steady temperature profile with $u = 2$ and $v = 1$. Boundary conditions: $T = 0$ except for the first 12.5% of the x-axis. Note the excessive numerical diffusion of the HYBRID scheme and the overshoots of the QUICK and Second order upwind schemes. Mesh: $40x40$.

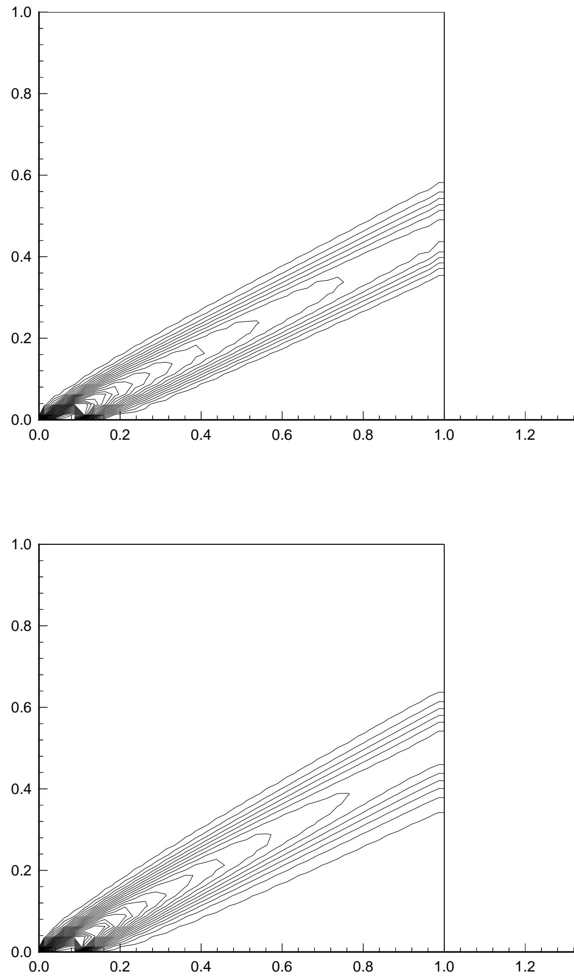

Figure 4: Comparison between QUICK (top) and Second order upwind (bottom) scheme, both limited with the Van Leer limiter. The Second order upwind scheme with the Van Leer limiter is usually referred to as the Van leer scheme. Due to the discontinous nature of the limiter, these schemes suffer from convergence problems, and one cannot generally expect better convergence than about 0.5%.

where index n indicates neighboring nodes. Now, the Scarborough criterion states that for a Gauss-Seidel like linear solver to converge the following expression must be satisfied for all control volumes in the field:

$$\frac{\sum_n |a_n|}{|a_p|} \leq c \tag{19}$$

where c is a constant depending on the iteration scheme and the coefficients. Loosely speaking, the more implicit the iteration process is, the larger we may choose c. For our purpose, it is desired that this inequality is fulfilled with $c \sim 1$.
This is generally not the case for a higher order convective discretization, and we therefore emply the following adaptive under-relaxation with Φ_{po} being the value of Φ_p at the previous iteration.

$$\alpha = \sum_n |a_n| - a_p \tag{20}$$

and

$$(a_p + \alpha)\,\Phi_p - \sum_n a_n \Phi_n = S_u + \alpha \Phi_{po} \tag{21}$$

We note that the scarborough criterion is fulfilled with a_p replaced with $a_p + \alpha$ *if* the right hand side is constant, that is if Φ_{po} is never updated, and that when the solution has converged, $\Phi_{po} = \Phi_p$, which implies that equation (18) is fulfilled, giving a consistent formulation. If Φ_{po} is updated every iteration we instead have to fulfill the following criterion:

$$\frac{\sum_n |a_n| + \alpha}{|a_p + \alpha|} \leq c \tag{22}$$

If a_p is positive, the adaptive under-relaxation procedure will guarantee that the above criterion is fulfilled for $c \geq 2$ which seems sufficient for most iterations schemes applied with the SIMPLE algorithm. If this is not the case, several sweeps with the solver have to be made before updating Φ_{po}.

10.3 The second-order BDF time-discretization scheme

A linear multistep time-discretization scheme may be written

$$h_1 \frac{\partial \Phi}{\partial t} = \beta_1 \Phi_1 + \beta_2 \Phi_2 + \beta_3 \Phi_3 \tag{23}$$

For the BDF scheme of order 1 (Backward Euler) β_1 is 1 , β_2 is -1 and β_3 is 0. We will now derive the coefficients for the second order BDF scheme:

$$\left\{ \begin{array}{rcl} \Phi_2 & = & \Phi_1 - b_1 h_1 - b_2 h_1^2 \\ \Phi_3 & = & \Phi_1 - b_1 (h_1 + h_2) - b_2 \left(h_1 + h_2\right)^2 \\ h_1 \left.\dfrac{\partial \Phi}{\partial t}\right|_1 & = & h_1 b_1 \end{array} \right. \tag{24}$$

Now eliminating b_2 by gaussian elimination we get

$$b_1 \left(h_1 + h_2\right) - b_1 h_1 \frac{\left(h_1 + h_2\right)^2}{h_1^2} = -\Phi_3 + \Phi_1 + \left(\Phi_2 - \Phi_1\right) \frac{\left(h_1 + h_2\right)^2}{h_1^2} \tag{25}$$

$$b_1 \left(h_1^2 \left(h_1 + h_2\right) - h_1 \left(h_1 + h_2\right)^2\right)\right) = -\Phi_3 h_1^2 + \Phi_1 h_1^2 + \left(\Phi_2 - \Phi_1\right) \left(h_1 + h_2\right)^2 \tag{26}$$

$$\tag{27}$$

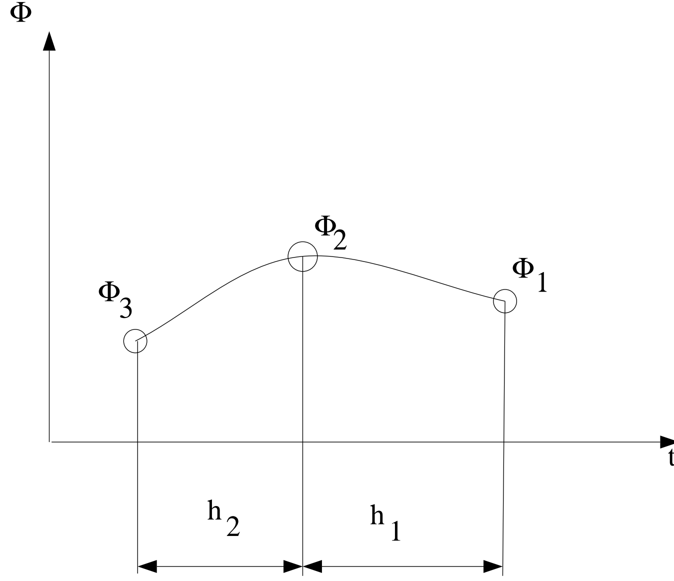

Figure 5: Nomenclature for second order BDF time discretization scheme.

and identifying the coefficients we get

$$\beta_1 = \frac{h_1^3 - h_1 (h_1 + h_2)^2}{h_1^2 (h_1 + h_2) - h_1 (h_1 + h_2)^2} = \tag{28}$$
$$= \frac{h_2 + 2h_1}{h_2 + h_1}$$

$$\beta_2 = \frac{h_1 (h_1 + h_2)^2}{h_1^2 (h_1 + h_2) - h_1 (h_1 + h_2)^2} = \tag{29}$$
$$= -\frac{h_1 + h_2}{h_2}$$

$$\beta_3 = -\beta_1 - \beta_2 \tag{30}$$

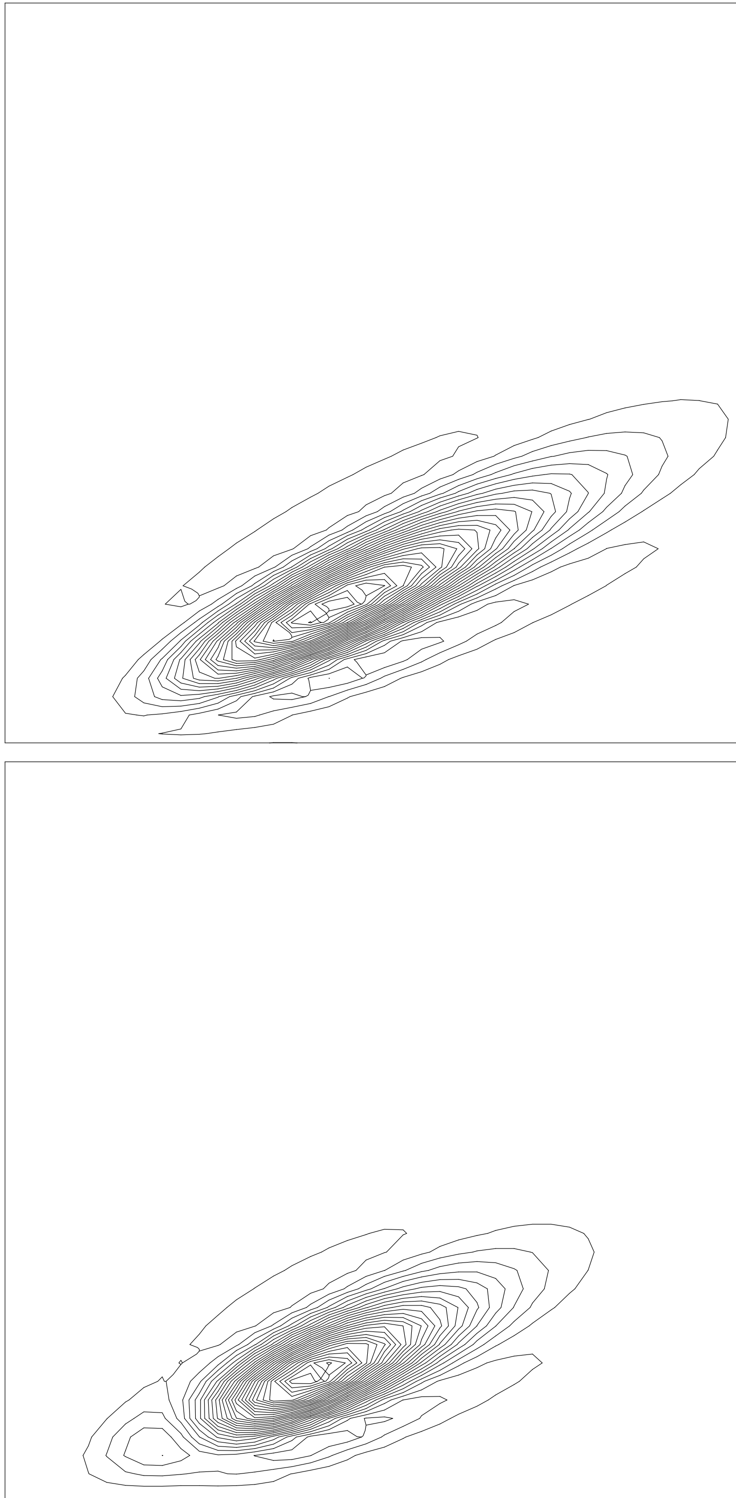

Figure 6: Travelling pulse of length 18% of the x-axis starting at the origin at $t = 0$. Here shown at $t = 0.25$ after 10 timesteps. $u = 2$, $v = 1$. Mesh: $40x40$. The QUICK scheme is used for space discretization. Backward Euler (top) and second order BDF (bottom) time discretization results. Note the smearing in the streamwise direction due to the first order Backward Euler scheme, as well as the undershoot behind the pulse in the BDF plot.

Index

action 7
adconv 11
aprc 14
base 27
Backward Euler scheme 12
BDF scheme 12, 31, 35
`bound` 28
`boundary` 17
boundary type 7
  given profile 7
  multiblock 7
  outflow profile 8
  symmetry 7
  wall 7
`bourd` 17
`calcp` 30
`calcp2` 30
`calcp3` 30
`cbase` 25
`chkbou` 26
`concom` 31
conin 30
conout 30
`conset` 30
consou 31
`correc` 30
cursor 20
`cweigh` 29
ddendt 19, 31
dtfmin 14
dtmin 14
dvoldt 31
extra 20
face 3
  negative 3
  positive 3
`fackey` 26
`facret` 21
`flowini` 13
fue 19
fuh 19
fun 19
fve 19
fvh 19
fvn 19
fwe 19
fwh 19
fwn 19
Gauss-Seidel 33
`gcent` 29
`getpp` 27
`getref` 27
`getres` 27
`getwin` 25
`ggeom` 29
ghost cell 3
ghost cell buffer 20
`gnowin` 25
hybrid scheme 12
icheck 13
ifsiz 4
imwin 4
index 3
  dependent 3
`initfa` 5, 26
`init` 31
initur 13
iomax 4
`ioval` 24
it 4
`jacobi` 28
jt 4
`key` 26
kt 4
layer1 20
layer2 20
layer buffer 20
ma1 20
ma2 20
maxbl 4
mcrit 13
`mdcon` 30
mmesh 5, 19, 31
msiz 4, 20
mulfac 30
`multimesh` 16
`mvmesh` 31
ngrid 5
`nmmax` 27
`nomove` 19, 31
pressure reference 11
QUICK scheme 12, 30, 32
`rdmesh` 16
refnod 11
`relax` 30
`rface` 26
`sbound` 27
Scarborough criterion 33

sendpp 27
setbou 8
setio 10
setref 27
setres 27
setwin 6, 25
sface 25
sind 21
snowin 5, 25
store 20
tface 22
totout 30
turini 13
twodim 5
ucoef 30
upcoef 30
upconv 30
urfmin 14
use 13
Van Leer scheme 12
wall functions 29
wallf 29
whtodo 7
window 4
xtrapl 23
xxtra 24
zerres 27