In this file we collect sketches, drawings and explanations
for the WABBIT code

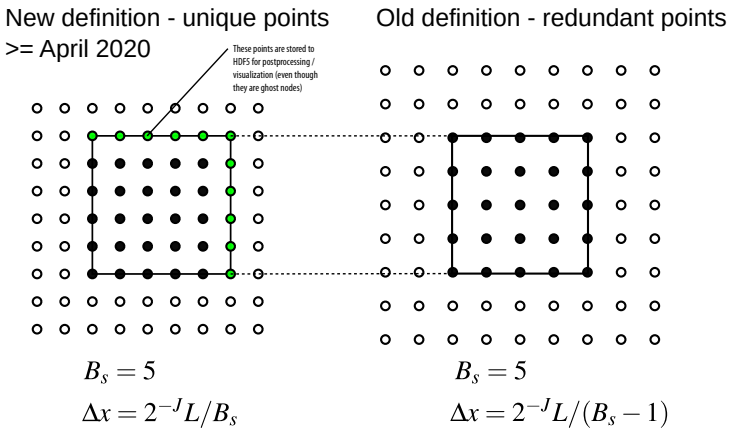
https://github.com/adaptive-cfd/WABBIT

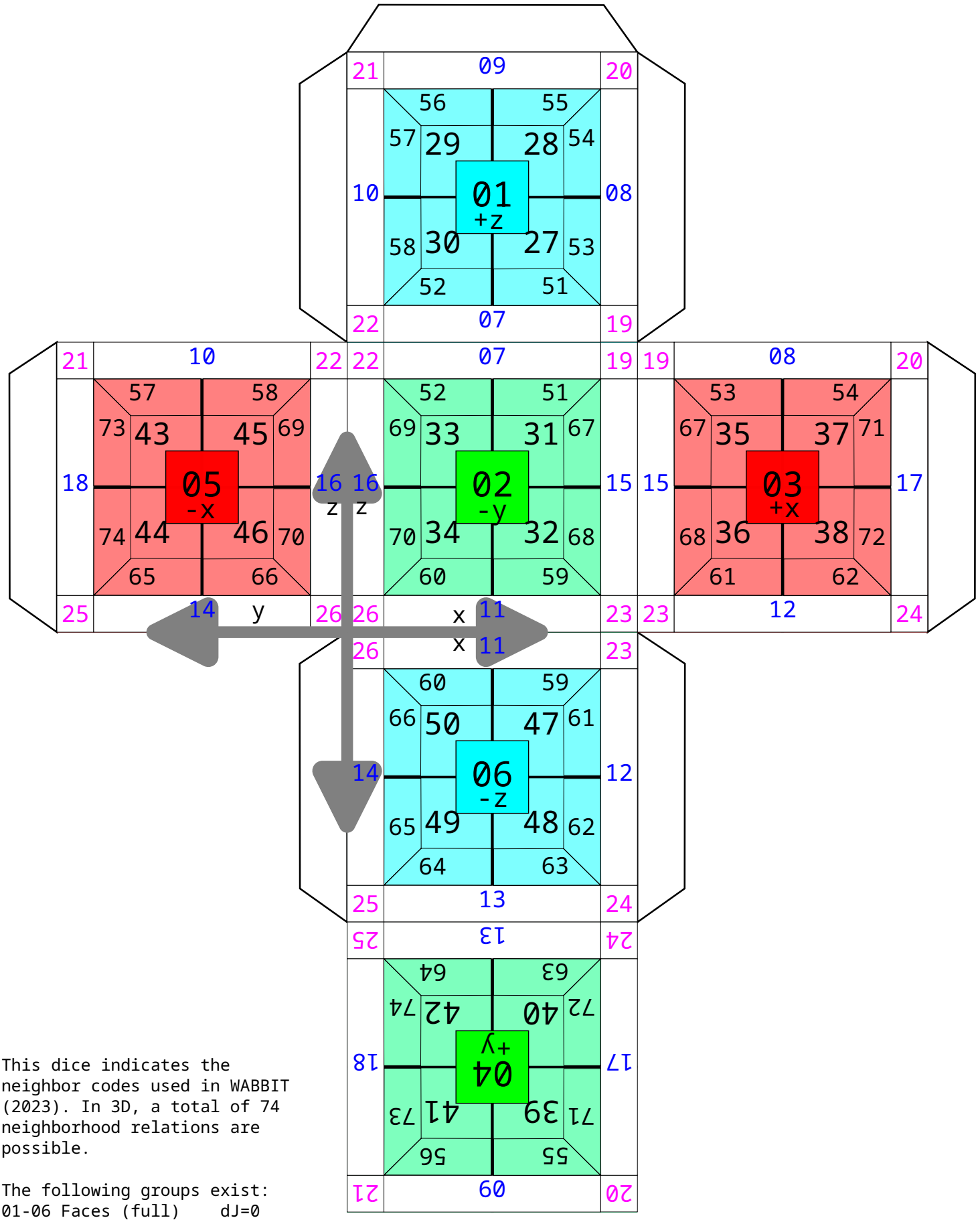
Version of WABBIT:
WABBIT v2.0beta5
06 may 2024, main branch, former "newBiorthogonal"

Authors:

T. Engels, CNRS & Aix-Marseille Université, thomas.engels@univ-amu.fr

Definition of a block in current version (uniqueGrid) and old version (redundantGrid)



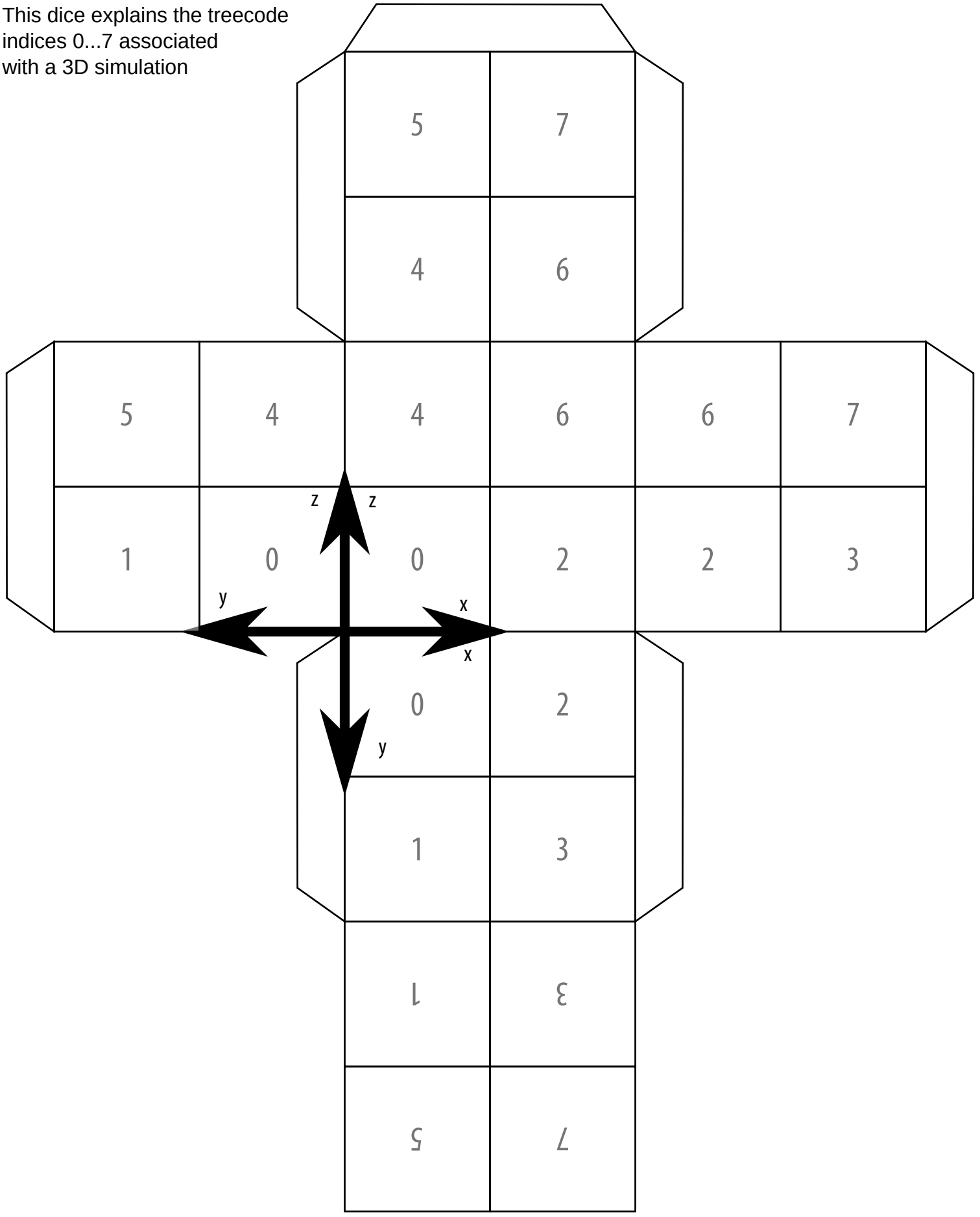


This dice indicates the neighbor codes used in WABBIT (2023). In 3D, a total of 74 neighborhood relations are possible.

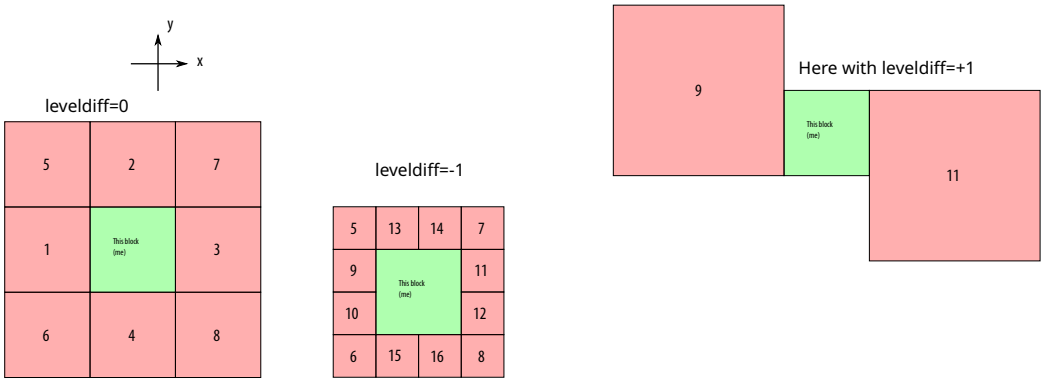
The following groups exist:
01-06 Faces (full)   dJ=0
07-18 Edges (full)   dJ=0
19-26 Corners (all)   dJ={-1,0,+1}
27-50 Faces (partial) dJ={-1,+1}
51-74 Edges (partial) dJ={-1,+1}

dJ = -1 Neighbor is finer (dJ == leveldiff)
dJ = 0 Neighbors have same resolution
dJ = +1 Neighbor is coarser

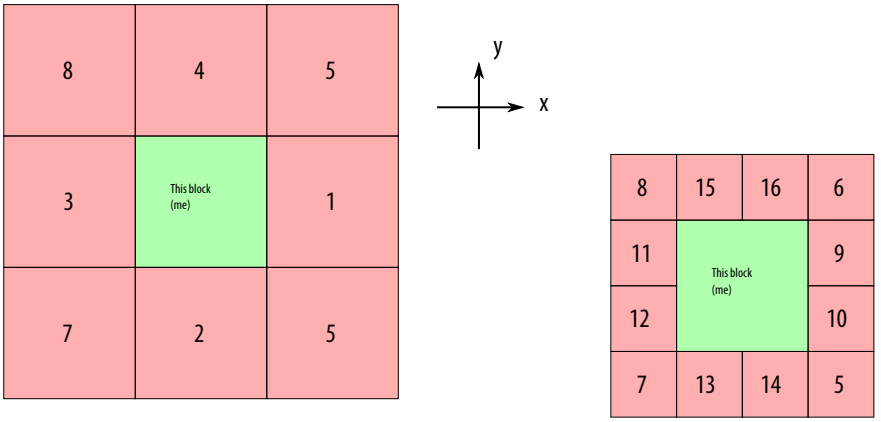
This dice explains the treecode indices 0...7 associated with a 3D simulation



Neighborhood relations in 2D
With each direction (e.g: top left) we have an associated code (dir)
This graph shows which code corresponds to what direction
For 3D please use the neighbor dice

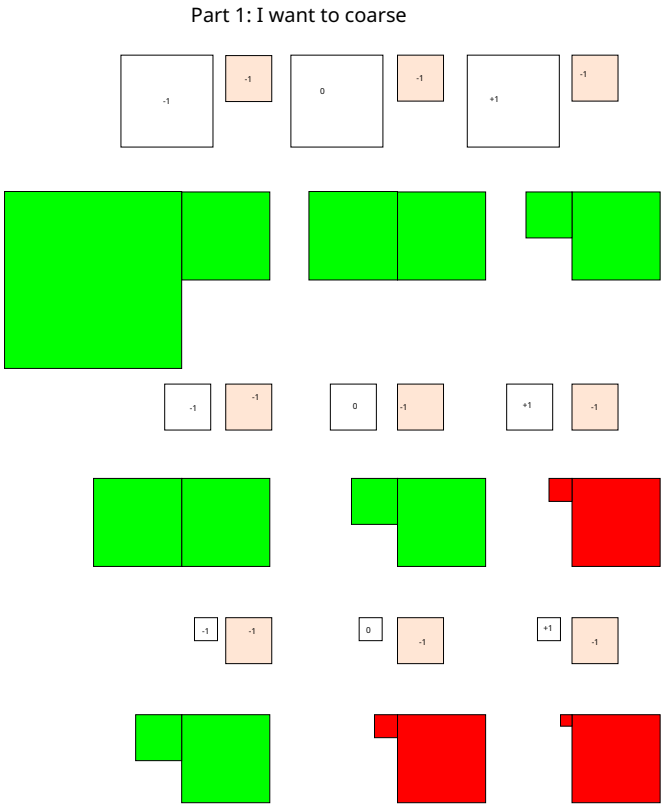


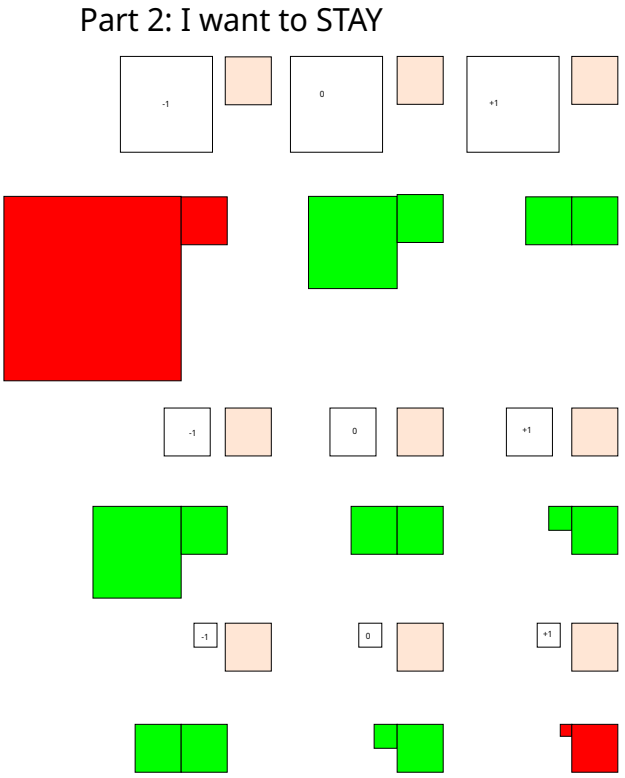
This is the coding used in HVY_NEIGHBORS
In the ghost node modules, it corresponds to SENDER

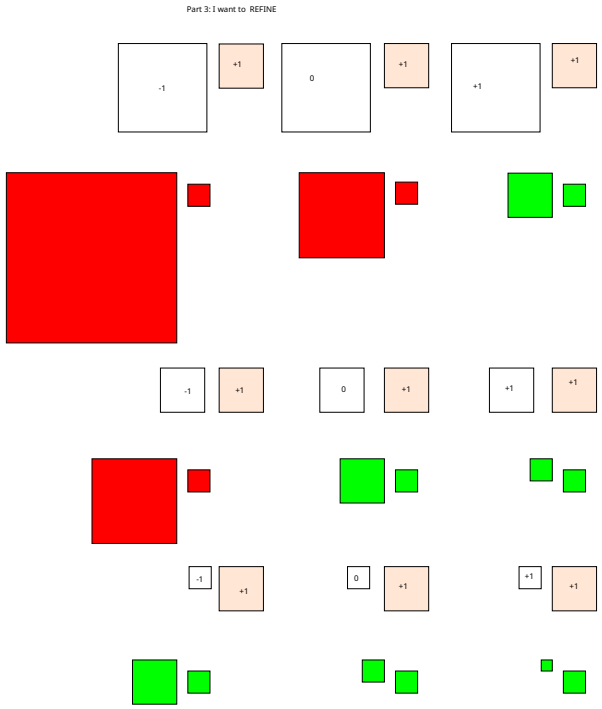


This is the INVERSE coding used for the RECEIVERS

Gradedness figures
These sketches were used to figure out the condition tree in ensure_gradedness.
They are drawn in 2D bbut apply exactly the same (independent of the neighbor code)
in 3D as well. The Orange block is "me" and the white one is
te neighbor. The green and red figures are the situation if both blocks do what their flag indicates,
Red means the situation is not valid, and hence flags have to be changed.



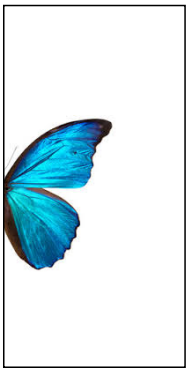




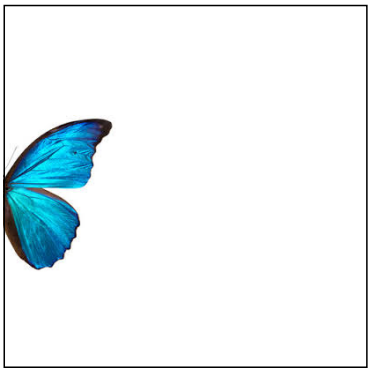
Symmetry condition ATTENTION AS OF 05/2024 THIS IS NOT IMPLEMENTED in the new version

Full simulation, not exploiting lateral symmetry

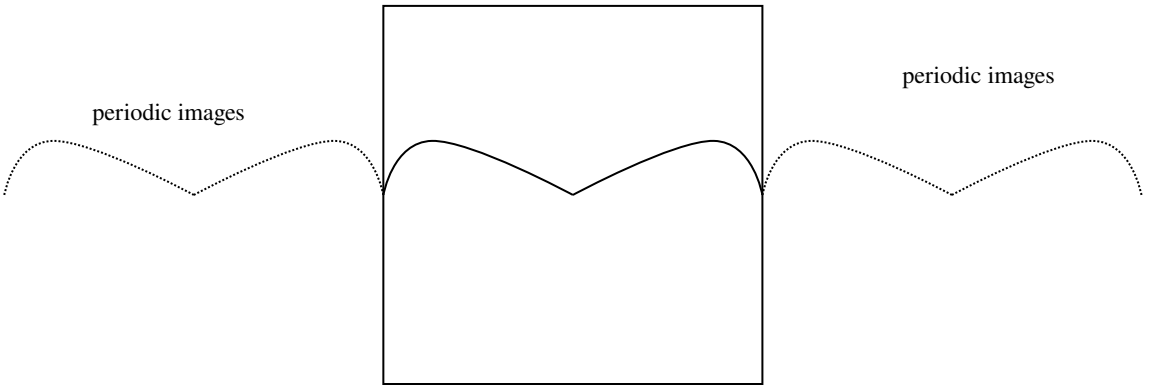
Ideal setup with rectangular domain exploiting symmetry



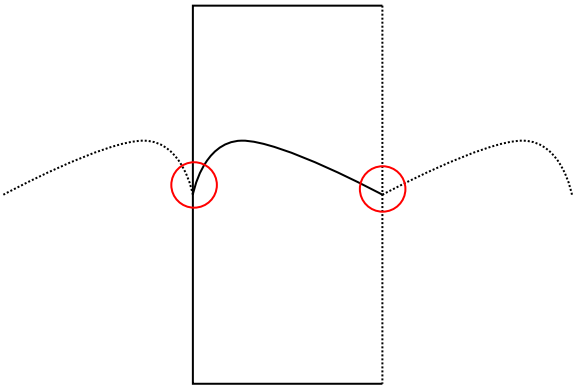
How a simulation looks like with symmetry_BC(2)=.true.



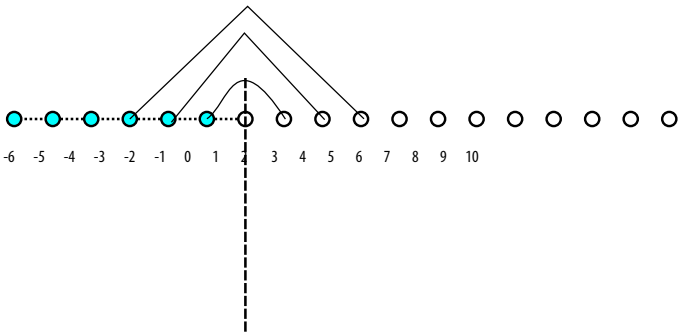
periodic, symmetric problem



we wish to simulate only half the domain



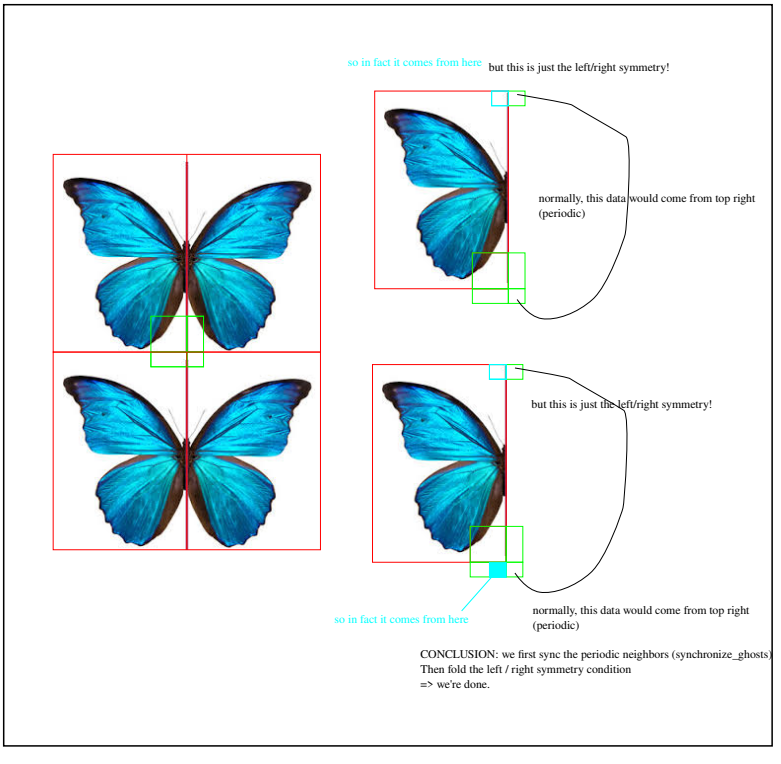
in both cases, the ghost node layer is simply filled with values from inside the block, in reverse order



There is two locations where this change will appear:

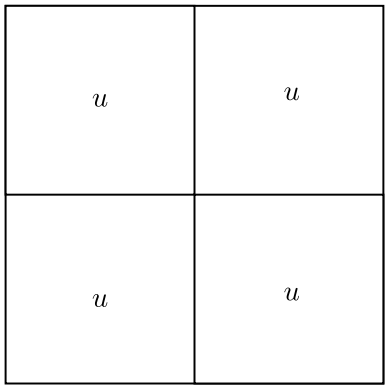
* when setting up neighbors, we must remove the neighbor code from non-existing neighbors. this causes the ghost node sync'ing routine to skip these borders
the point is the redundant nodes right on the domain border: they must not be synch'ed !

* filling the ghost node layer itself, i.e., the local copy action (no MPI here!) Attention: interferes with INTERPOLATION because those patches extend to the ghost nodes layer

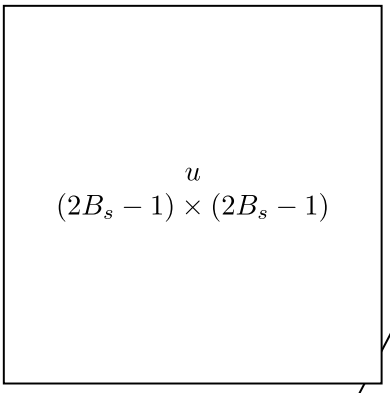


Thoughts for the complete wavelet transform
Those sketches are correct but this part is not yet implemented
At the moment, we do FWT / IWT only on ONE block.

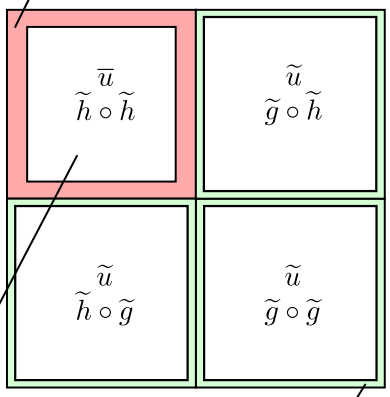
Transform acts on 4 (2D) /8 (3D) blocks



Virtual, merged block
(same resolution)
Transform deals with this data



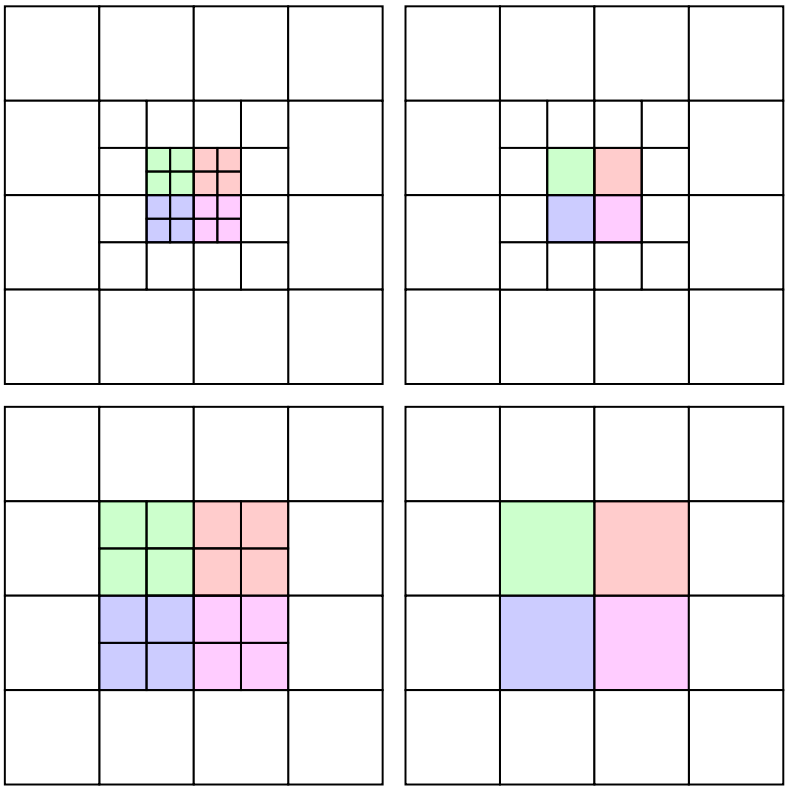
wrong scaling function coeffs
(width: support h)
According to N Kevlahan, copy fine data



"Mallat-order"

copied to a coarser, virtual
block (scaling function, coarser approx.)

wrong wavelet coeffs
(width: support g)



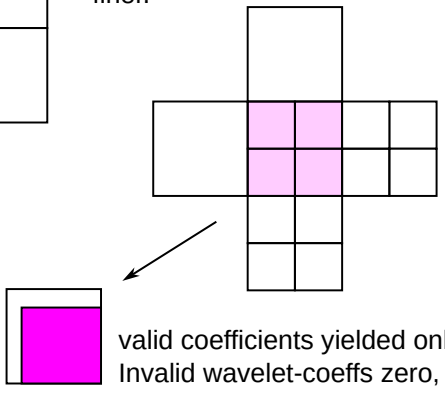
Note: errors occur only
at fine/coarse interfaces

If g=6, then those errors
disappear at block interfaces
on same level:

Notes:
(i) this means that ghost
node sync'ing is required in the
process, but only on level J and
only for "same" neighbors

(ii) it requires an interface to detect "same
level neighbors".

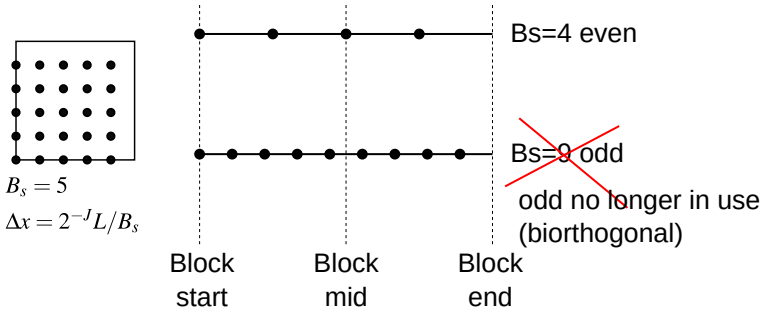
(iii) transform proceeds level-wise from fine
to coarse. drôlement, the always means
neighbors either same/coarser, but never
finer.

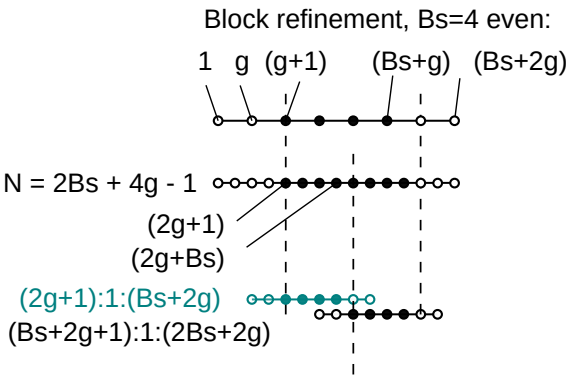


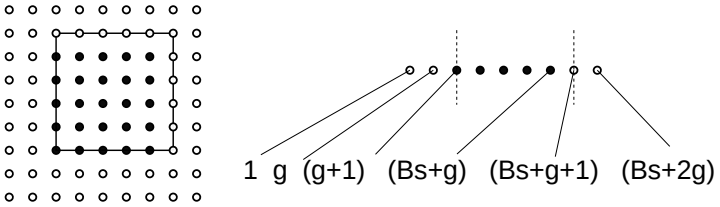
valid coefficients yielded only in purple region.
Invalid wavelet-coeffs zero, scaling function
coeffs obtained as decimation of fine signal.

$\widetilde{h}\circ\widetilde{h}\,\overline{u}$ $\widetilde{g}\circ\widetilde{h}\,\widetilde{u}$

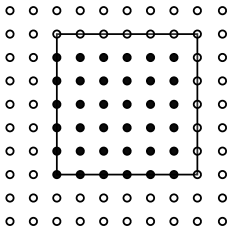
Grid A: unique grid







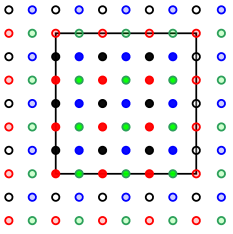
Block view



Bs=6, g=2

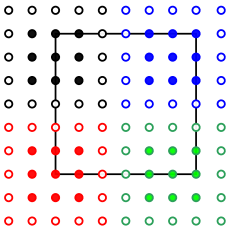
WC in spaghetti ordering

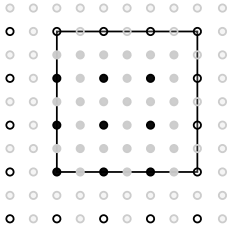
- SC [1: Bs+2g-1 : 2 | 1: Bs+2g-1 : 2]
- WCx [2: Bs+2g : 2 | 1: Bs+2g-1 : 2]
- WCy [1: Bs+2g-1 : 2 | 2: Bs+2g : 2]
- WCxy [2: Bs+2g : 2 | 2: Bs+2g : 2]



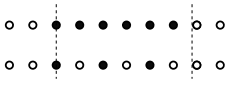
- SC [2: Bs+2g : 2 | 2: Bs+2g : 2]
- WCx [3: Bs+2g-1 : 2 | 2: Bs+2g : 2]
- WCy [2: Bs+2g : 2 | 3: Bs+2g-1 : 2]
- WCxy [3: Bs+2g-1 : 2 | 3: Bs+2g-1 : 2]

N = (Bs+2g)/2



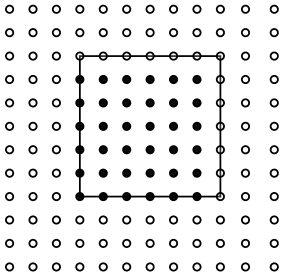


Inflated Mallat: 4 (or 8) times as many data
including many zeros, same representation for WC/SC.
Indices are the same as for SC in spaghetti ordering

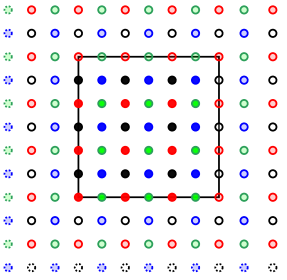


u: (g+1):(Bs+g)

wc/sc (Mallat-order)
(g+1):(Bs+g-1):2

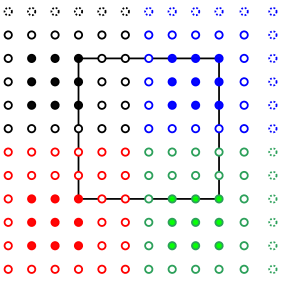


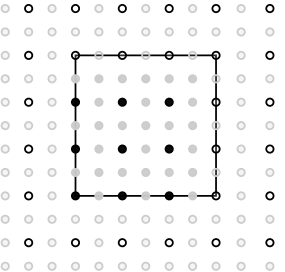
Bs=6, g=3



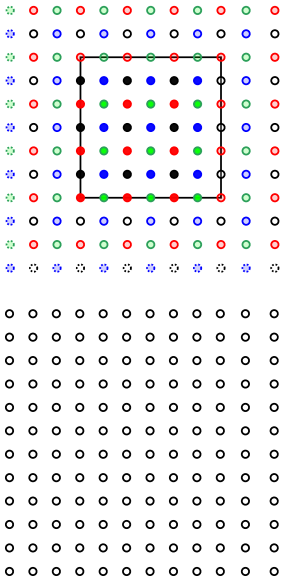
N = (Bs+2g)/2

Attention: interior
nodes become irregular.
Simple: a half-cell of periodic data
are taken (skipping two types of coefficients)

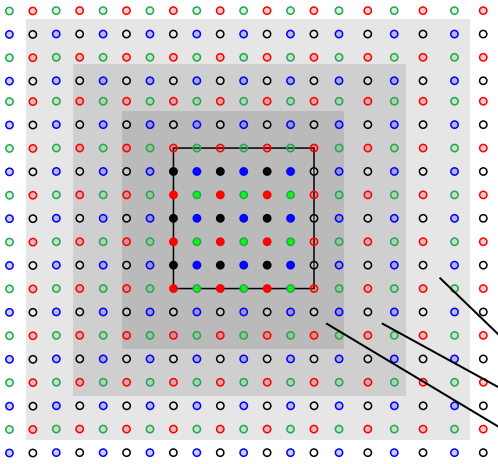




Bs=6, g=3



Bs=6, g=7



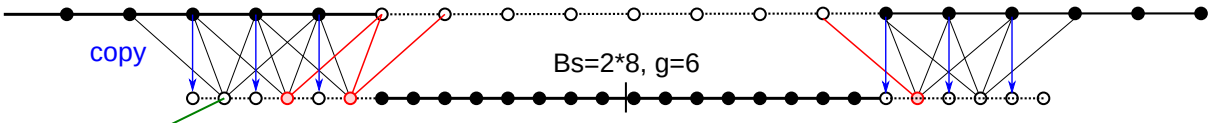
sync'ing of spaghetti-ordered coefficients

CDF44 g_eff = 6
CDF42 g_eff = 4
CDF22 g_eff = 2

for CDF44, we need 3 coeffs to either side

Notes on Ghost nodes

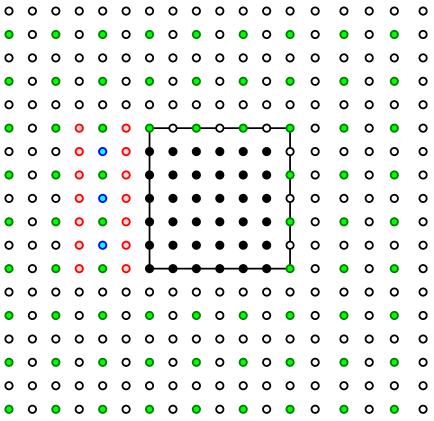
Idea: non-stage sync'ing



correctly interpolated

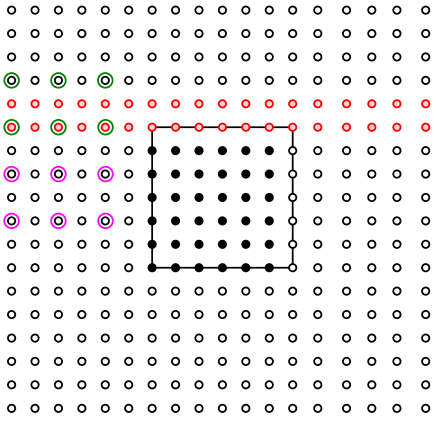
These 2 / 1 points can only be interpolated when the coarse blocks ghost nodes are filled. This is okay: we just fill them with garbage, then correct the interpolation on the finer block.

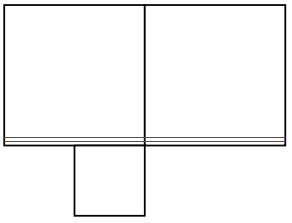
Bs=6, g=6



• copied from coarse

Red points to be re-interpolated
(or copied/unchanged if they match the coarse grid)
This curses the idea: the outermost red points cannot be interpolated locally on the fine block alone





Those lines cannot be interpolated w/o filling ghosts first (stage 1)

maybe it works if we set the layers to 0 first, and add on the fine block only what was missing on the coarse one?
Interpolation is a linear operation, we can exploit that.

On coarse: interpolate with all points from fine set to 0
On fine (correction): add to the exitsing value the interpolation with all coarse =0
The sum is the exact solution.

In 2D this saved 5% on my machine 4 cores 13000 blocks -> maybe not worthwhile

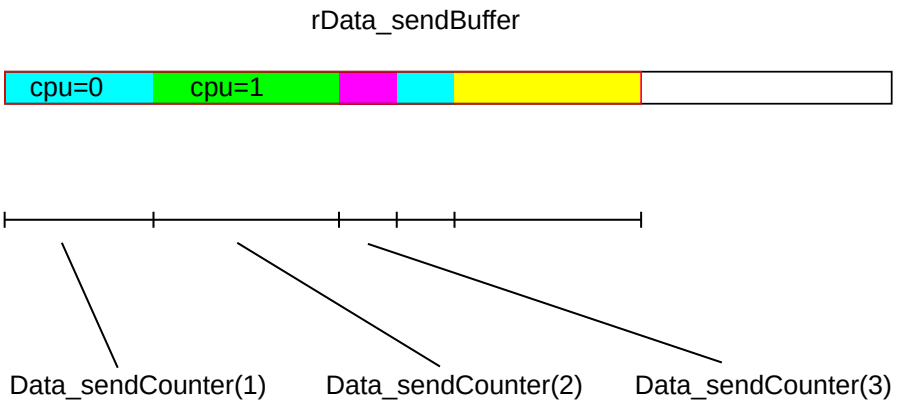
Ghost nodes notes

rData_sendBuffer
rData_recvBuffer
iMetaData_sendBuffer
iMetaData_recvBuffer

MetaData_sendCounter
MetaData_recvCounter

Data_sendCounter
Data_recvCounter

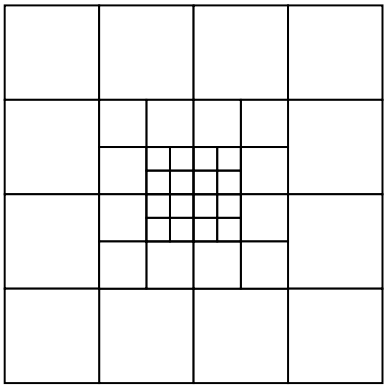
For example:

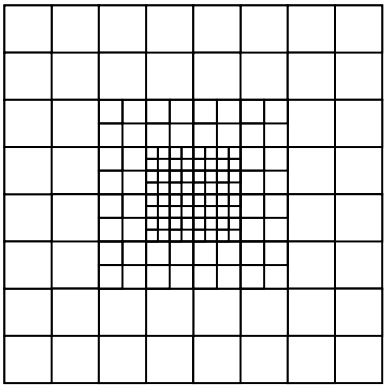


buffer for a partner mpirank:
ii0 = sum(MetaData_sendCounter(0:(k-1)-1)) + 1
ii1 = ii0 + MetaData_sendCounter(k-1)-1
iMetaData_sendBuffer(ii0:ii1)

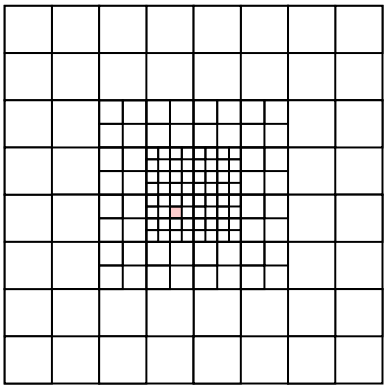
Concept of safety & security zone (we need better words)

Safety-zone: refine everywhere, before evolving in time

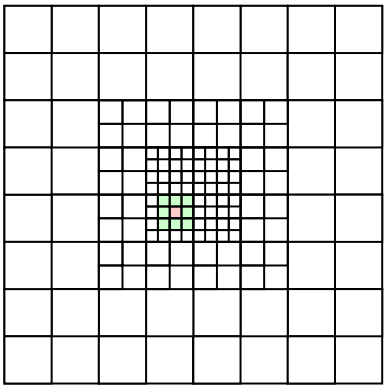




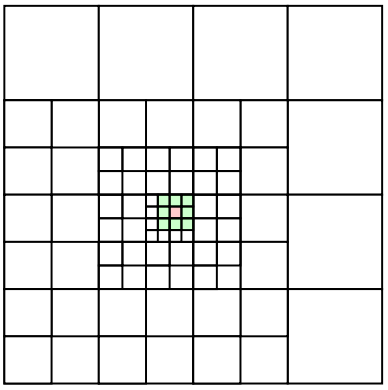
Security zone: keep blocks adjacent to significant blocks, in order to avoid significant details in the coarseExtension zone



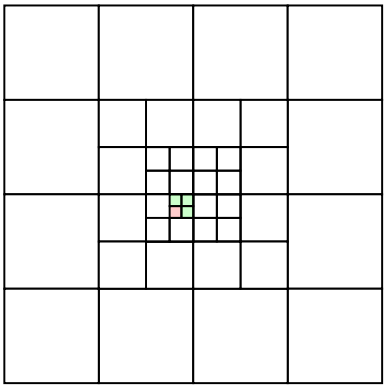
say 1 block is significant



then we will keep the security blocks

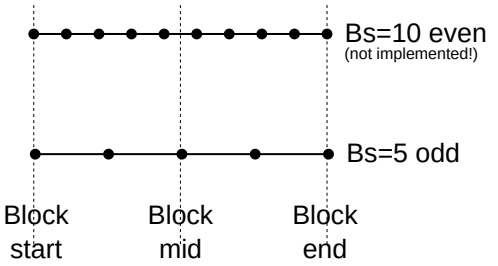


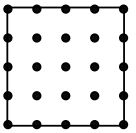
and coarsen what we still can



this would be the grid w/o security zone

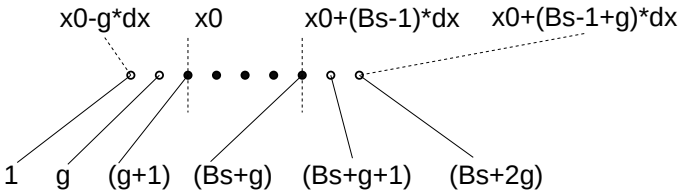
Grid B: redundant grid



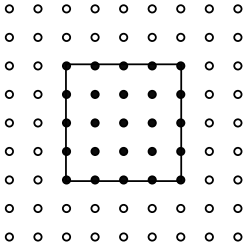


$B_s = 5$

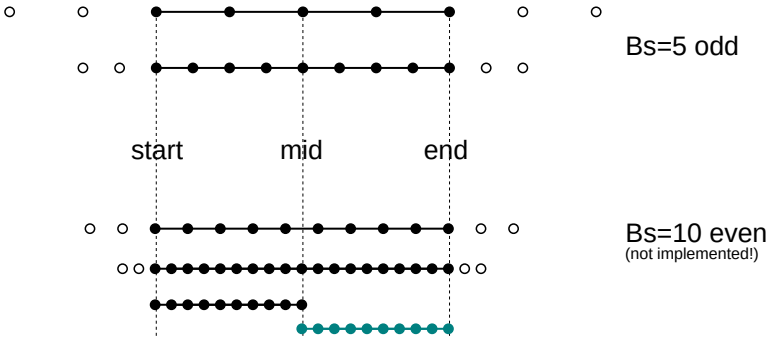
$\Delta x = 2^{-J} L/(B_s - 1)$



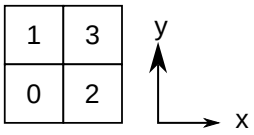
Complete 2D block layout:

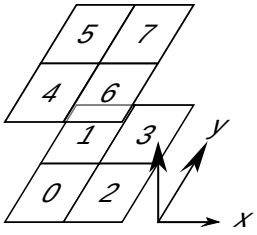


Refinement / Splitting

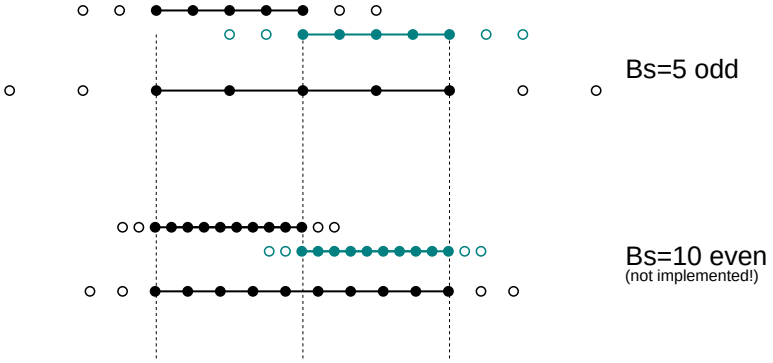


Treecodes:

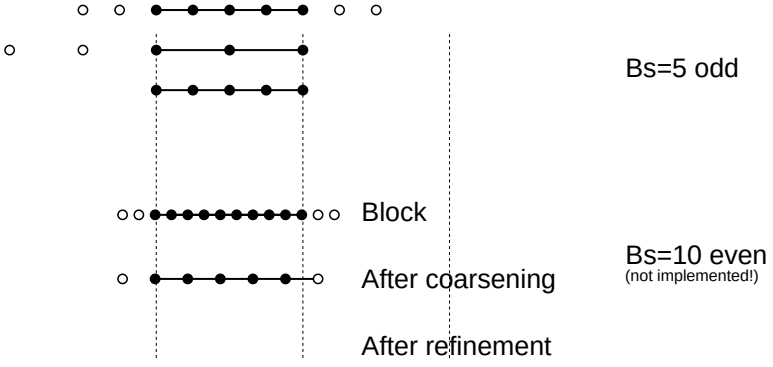




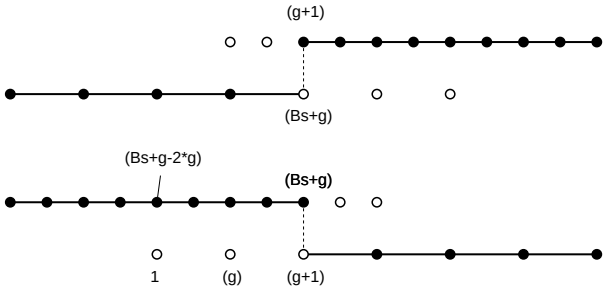
Coarsening / Merging



Thresholding (refine(coarsen(block))



Coarse/fine interface (1D)

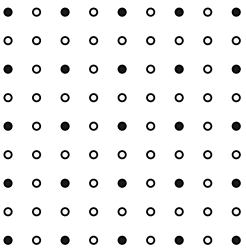


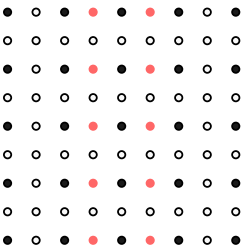
see interpolation_testbed.m

interpolation (2D)

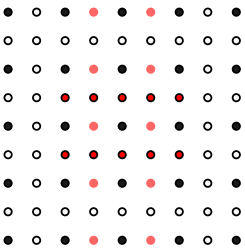
here for 4th order stencil

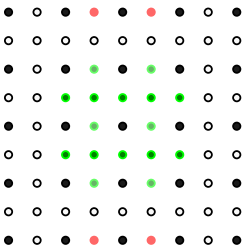
interpolated points





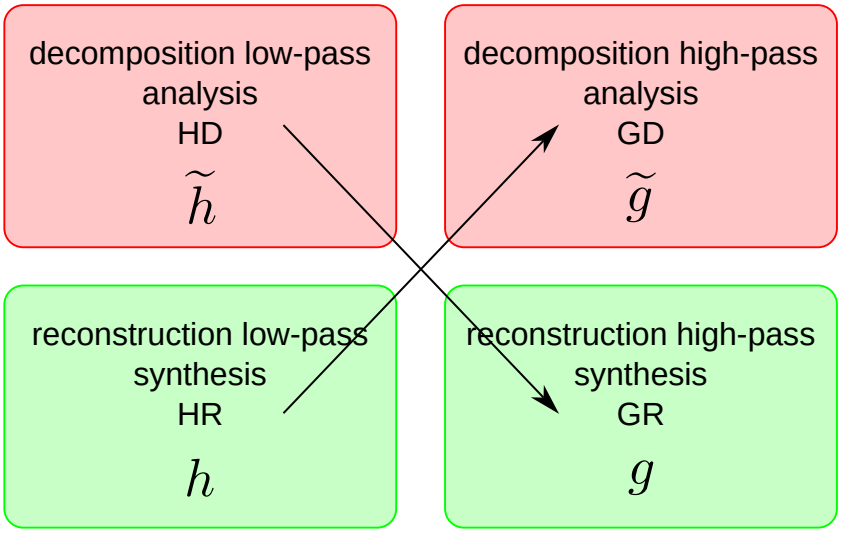
```
! interpolate regular columns
do ixfine =  4, nxfine-3, 2
    fine( ixfine, 1:nyfine:2 ) = &
    b(2)*fine( ixfine-3, 1:nyfine:2 ) &
    + b(1)*fine( ixfine-1, 1:nyfine:2 ) &
    + b(1)*fine( ixfine+1, 1:nyfine:2 ) &
    + b(2)*fine( ixfine+3, 1:nyfine:2 )
enddo
```





```
do iyfine =  4, nyfine-3, 2
    fine( 1:nxfine, iyfine ) = b(2)*fine( 1:nxfine, iyfine-3 ) &
    + b(1)*fine( 1:nxfine, iyfine-1 ) &
    + b(1)*fine( 1:nxfine, iyfine+1 ) &
    + b(2)*fine( 1:nxfine, iyfine+3 )
enddo
```

Biorthogonal wavelets - filters



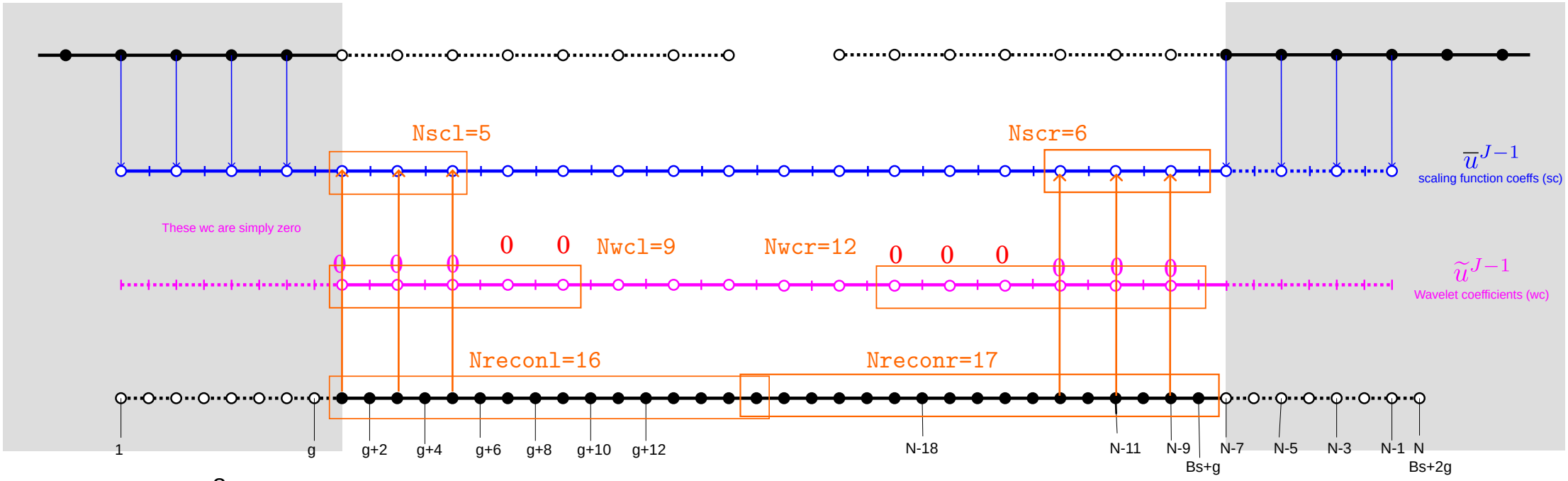
$$\widetilde{g}_i = (-1)^i \, h_{-i+1}$$
$$g_i = (-1)^i \, \widetilde{h}_{-i+1}$$

$$\widetilde{g}_i = (-1)^i \, h_{-i+1}$$

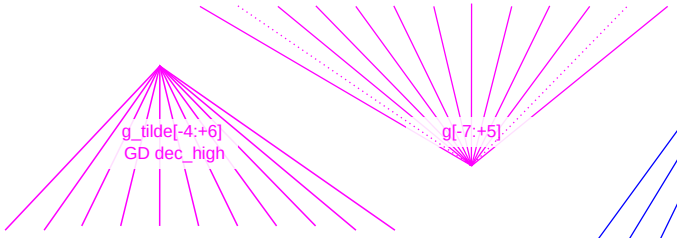
CDF62

zeros: I

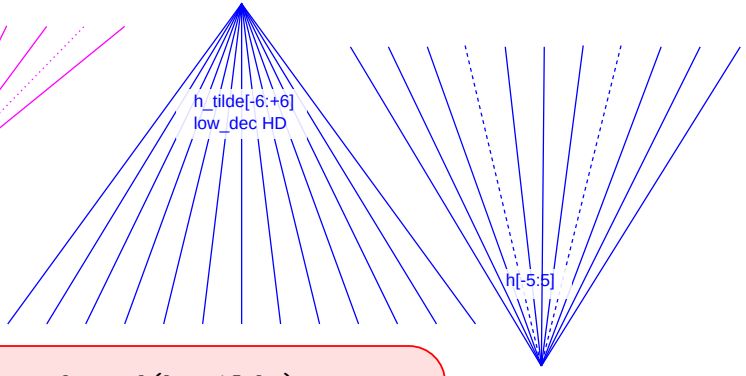
Note: these drawings are WITHOUT redundant point



g=8



Bs=32




```
Nscl = lbound(h_tilde)-1        Nscr = ubound(h_tilde)
Nwcl = Nscl + lbound(g_tilde)   Nwcr = Nscr + ubound(g_tilde)
Nreconl = Nwcl + lbound(g)      Nreconr = Nwcr + ubound(g)
```

Nscl, Nscr:
determined with (h_tilde filter) not reaching into ghost nodes.
Remaining points are copied (orange arrow)

Pink zero wc: reconstruction of last ghost node with (g filter)
must use all zero wc (YOU CAN SKIP THIS!)

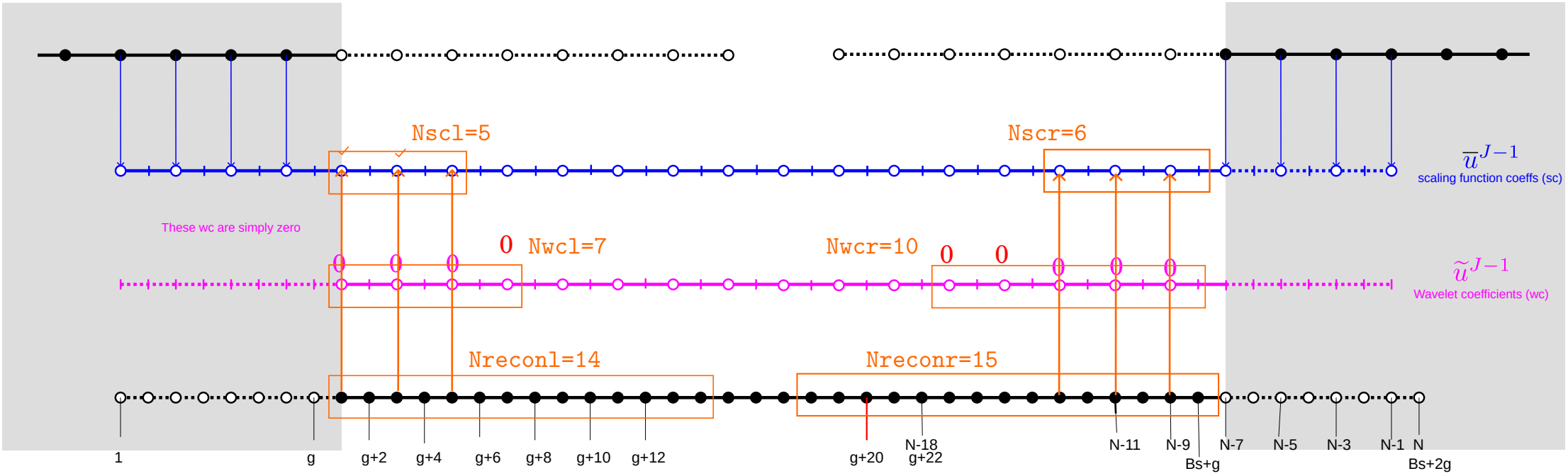
Red zero WC: wc that is computed from copied SC is zero
(g_tilde filter)

Reconstruction: any modified zero or copy (g filter & h filter)
as the g filter is wider, it determines the length (but check!)
-> it is always the g filter because it is wider AND we set more WC to zero

CDF44

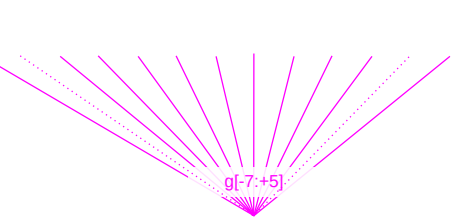
zeros: I

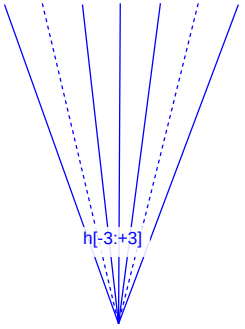
Note: these drawings are WITHOUT redundant point

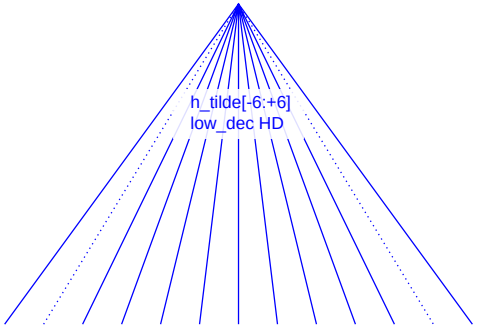


g=8

Bs=32

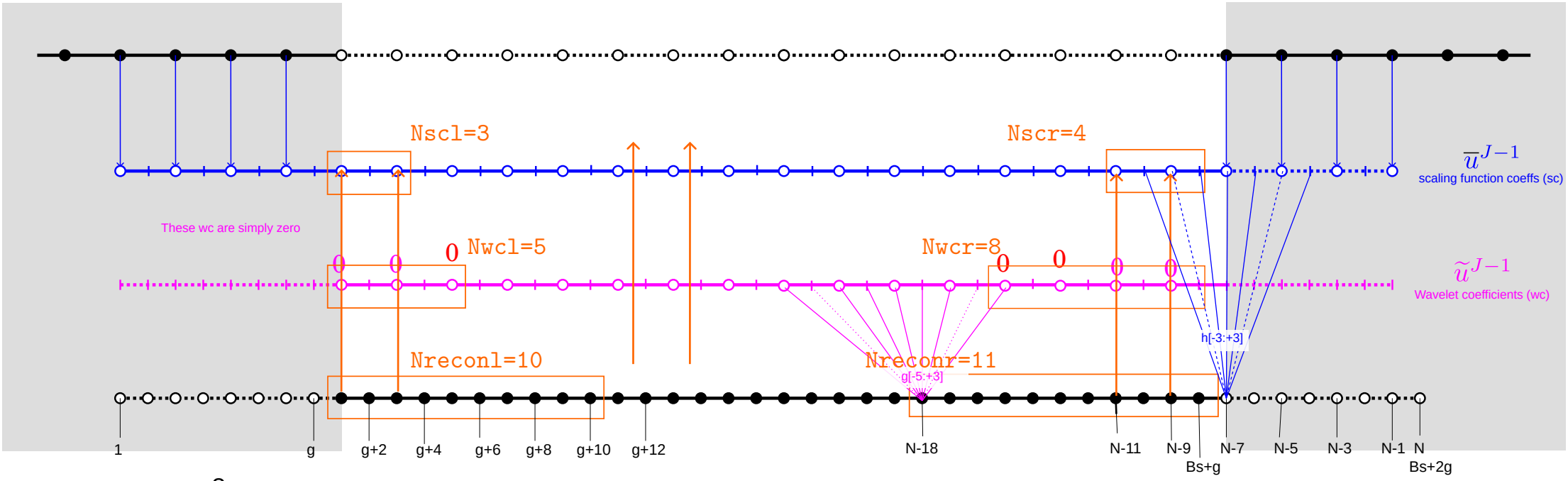






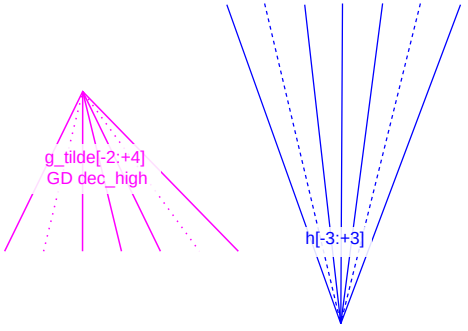
CDF42

Note: these drawings are WITHOUT redundant point



g=8

Bs=32

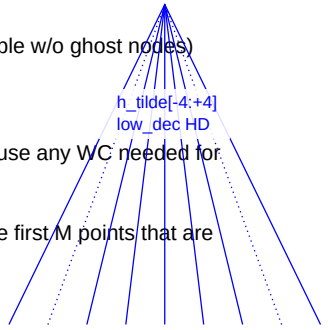


Step 1: gather SC on coarse grid.. Fine block computes some (which are possible w/o ghost nodes) and copies some. Send to coarse block
- ghost nodes on coarse block are done

Step 2: Coarse block interpolates fine block ghost nodes. This is possible because any WC needed for this are assumed zero. Now fine ghost nodes are filled

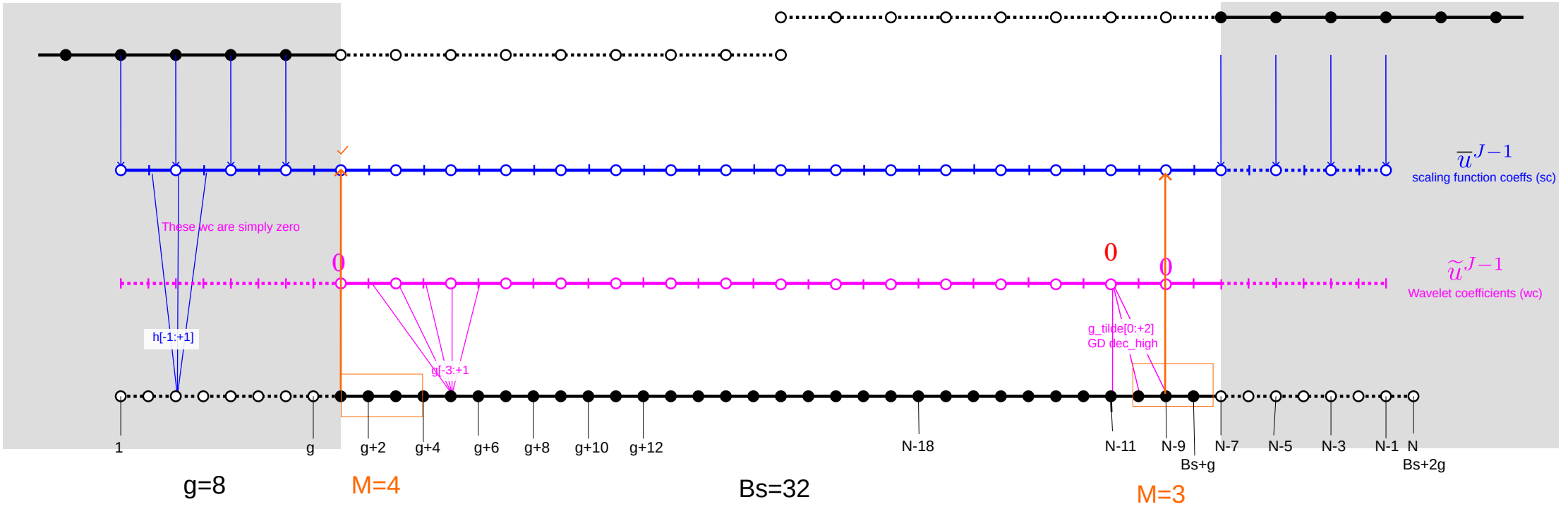
Step 3: substitution step. Fine block has coarse SC and WC available. Compute first M points that are affected by altered WC and SC

Gillis & Rees likewise include this substitution in the ghost node sync



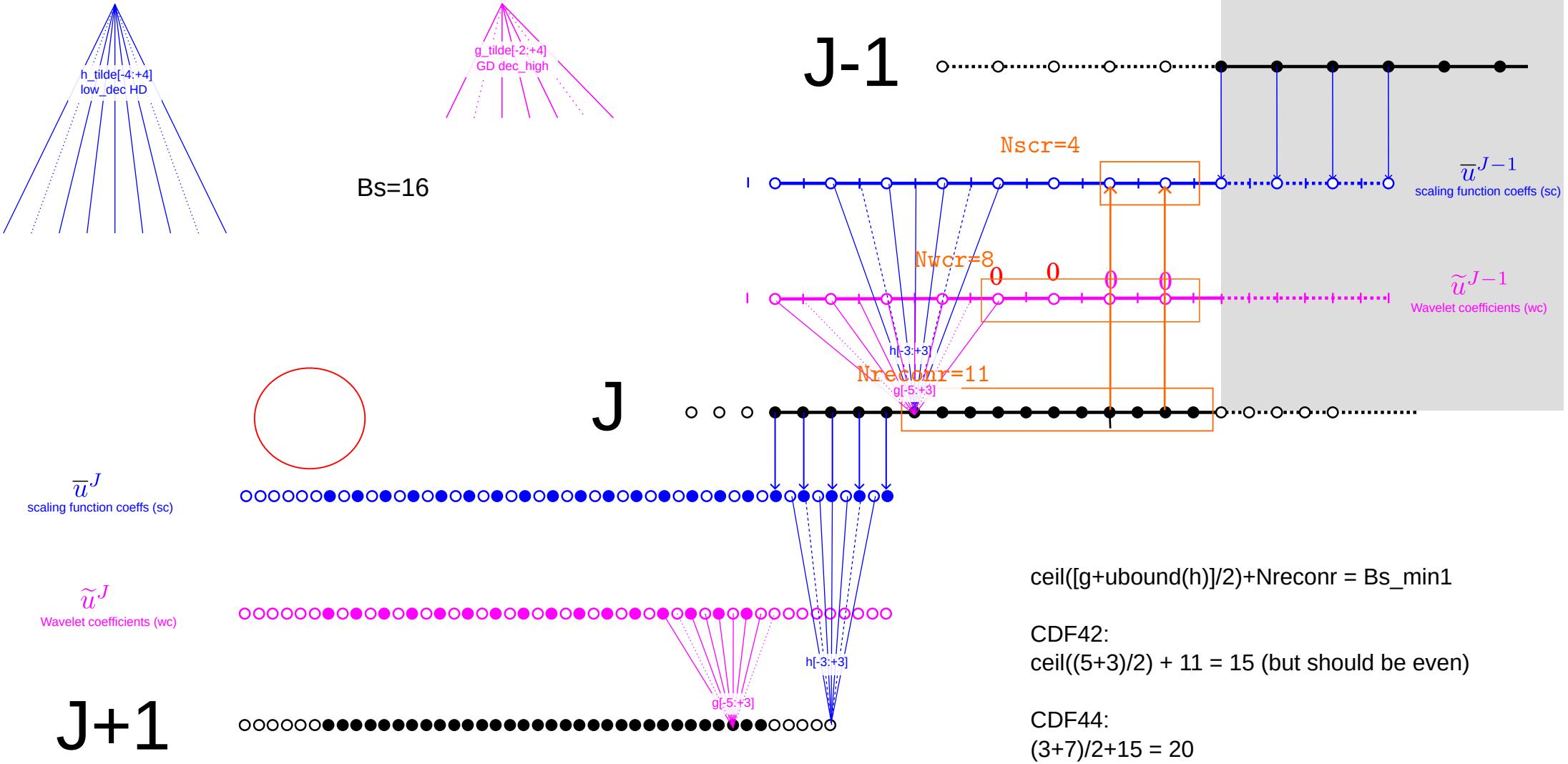
CDF22

Note: these drawings are WITHOUT redundant point



CDF42

Note: these drawings are WITHOUT redundant point



ceil([g+ubound(h)]/2)+Nreconr = Bs_min1

CDF42:
ceil((5+3)/2) + 11 = 15 (but should be even)

CDF44:
(3+7)/2+15 = 20

CDF62:
(7+5)/2+17 = 23