Appendix A

Implementation of MG parts

A.1 Restriction

The multigrid method (MG) has several grids of different resolution, and we need to convert the problem between the diffrent grids during the overarching the MG-algorithm. The restriction algorithm has the task of translating from a fine grid to a coarser grid. In this implementation we use a half weight stencil to restrict a quantity from a fine grid to a coarse grid. To get the coarse grid value it gives half weighting to the fine grid point corresponding directly to the coarse grid point, and gives the remaining half to the adjacent fine grid values, see (**??**), for 1D, 2D and 3D examples.

$$\begin{aligned}
\mathcal{R}_{1\mathrm{D}} &= \frac{1}{4}\begin{bmatrix}1 & 2 & 1\end{bmatrix} \\
\mathcal{R}_{2\mathrm{D}} &= \frac{1}{8}\begin{bmatrix}0 & 1 & 0\\ 1 & 4 & 1\\ 0 & 1 & 0\end{bmatrix} \\
\mathcal{R}_{3\mathrm{D}} &= \frac{1}{12}\left(\begin{bmatrix}0 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 0\end{bmatrix}, \begin{bmatrix}0 & 1 & 0\\ 1 & 6 & 1\\ 0 & 1 & 0\end{bmatrix}, \begin{bmatrix}0 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 0\end{bmatrix}\right)
\end{aligned} \tag{A.1}$$

In our implementaion we first cycle through all of the true coarse grid points, then the two main tasks is to find the specific fine grid point corresponding to the specific coarse grid point, and finding the indexes of the fine grid points surrounding the grid point.

Since the value in both grids are stored in a first order lexicographical array, we should treat the grid points in the same fashion, so the values are stored close to each other in the array. The first dimension is treated first, then the next is dimension is incremented followed by treating the first dimension again, then increment the next and so on. The fine grid has twice the resolution of the coarse grid, so for each time the coarse grid index is incremented, the fine grid index is

incremented twice.

Along the x-axis each incrementation is the number of values stored in the grid, which for scalars is 1 (This is only used for scalars, so now the 1 is hardcoded, I will need to test if using $size[0]$ instead affects speed), and 2 for the fine index. The fine index will in addition need to skip 1 row each time, each time the y-axis is incremented, due to the finer resolution and 1 layer each time the z-axis is incremented.

At the edges of the grid we have ghost layers, which have equal thickness for both the grids, so the coarse grid needs to increment over the ghost values, in the x-direction, each time y is incremented. When z is incremented the index need to skip over a row of ghost values. The fine index follows the same procedure as the coarse index when dealing with the ghost layers.

When correct fine grid index is found, corresponding to a coarse grid index, the stencil needs to be applied around that grid value. This is done by first calculating the index of the first coarse and find indexes and setting the correct indexes for the surrounding grid values, then the surrounding grid indexes can be incremented exactly as the fine grid index and they will keep their shape around the fine grid index. Since our indexes in x, y and z are labeled j,l,k, the next value along the x-axis is labeled 'fj' and the previous is labeled 'fjj'. The coarse and fine grid indexes are label 'c' and 'f' respectively.

```
//Indexes
long int c = cSizeProd[1]*nGhostLayers[1] +
    cSizeProd[2]*nGhostLayers[2] +
    cSizeProd[3]*nGhostLayers[3];

long int f = fSizeProd[1]*nGhostLayers[1] +
    fSizeProd[2]*nGhostLayers[2] +
    fSizeProd[3]*nGhostLayers[3];
long int fj  = f + fSizeProd[1];
long int fjj = f - fSizeProd[1];
long int fk  = f + fSizeProd[2];
long int fkk = f - fSizeProd[2];
long int fl  = f + fSizeProd[3];
long int fll = f - fSizeProd[3];
```

Listing A.1: Setting the stencil indexes

```
//Cycle Coarse grid
for(int l = 0; l<cTrueSize[3]; l++){
  for(int k = 0; k < cTrueSize[2]; k++){
    for(int j = 0; j < cTrueSize[1]; j++){
      cVal[c] = coeff*(6*fVal[f] + fVal[fj] + fVal[fjj] +
          fVal[fk] + fVal[fkk] + fVal[fl] + fVal[fll]);
      c++;
      f  +=2;
      fj +=2;
      fjj+=2;
      fk +=2;
      fkk+=2;
      fl +=2;
      fll+=2;
    }
    c  += cKEdgeInc;
    f  += fKEdgeInc;
    fj += fKEdgeInc;
    fjj+= fKEdgeInc;
    fk += fKEdgeInc;
    fkk+= fKEdgeInc;
    fl += fKEdgeInc;
    fll+= fKEdgeInc;
  }
  c  += cLEdgeInc;
  f  += fLEdgeInc;
  fj += fLEdgeInc;
  fjj+= fLEdgeInc;
  fk += fLEdgeInc;
  fkk+= fLEdgeInc;
  fl += fLEdgeInc;
  fll+= fLEdgeInc;
}
```

Listing A.2: The foor loop doing the calculations

As of now there is 2 seperate implementations, for 2 and 3 dimensions.

A.2 Prolongation

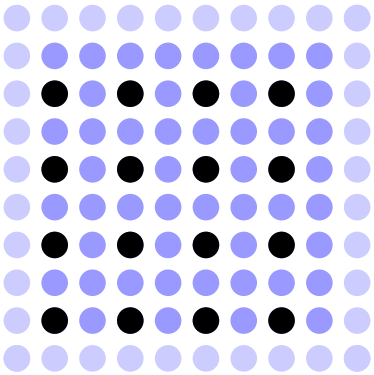
Along with the restriction operator described in the previous section, we also need prolongation operator to go from a coarse grid to a finer grid. Here we will use bilinear interpolation, for two dimensions and trilinear interpolation for 3 dimensions. In bilinear interpolation seperate linear interpolation is done in the x- and y-direction, then those are combined to give a result on the wanted spot. (Note to self: Add source here) The same concept is expanded to give trilinear interpolation. The two and three dimensional stencils is given in (**??**)

$$\begin{aligned} \mathcal{I}_{2\mathrm{D}} &= \frac{1}{4}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \\ \mathcal{I}_{3\mathrm{D}} &= \frac{1}{8}\left(\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 4 & 2 \\ 4 & 8 & 4 \\ 2 & 4 & 2 \end{bmatrix}, \begin{bmatrix} 2 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}\right) \end{aligned} \tag{A.2}$$

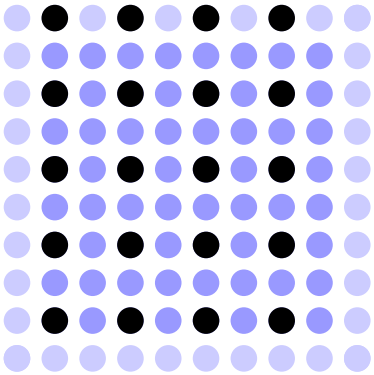
The algorithm implemented for the interpolation is based on the method, described in **??** has the following steps, which is also shown for a 2D case in **??**.

1. Direct insertion: Coarse-¿Fine
2. Interpolation on highest Dimension: $f(x) = \frac{f(x+h)+f(x-h)}{2h}$
3. Fill needed ghosts.
4. Interpolation on next highest Dimension

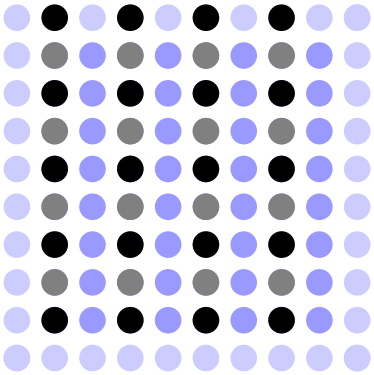
The interpolation should always first be done on the highest dimension, because the grid values are stored further apart along the highest axis in the memory, and the each succesive interpolation needs to apply to more grid points. (Note to self: should test)



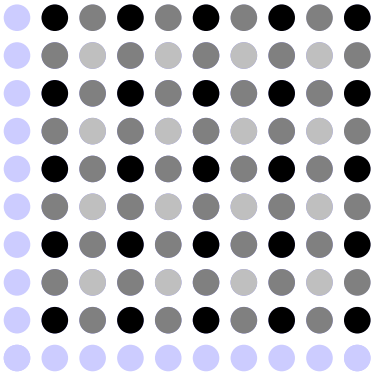
(a) Direct insertion



(b) Swapping X-Dim ghosts



(c) Y-swipe



(d) Swapping Y-ghostlayer and X-swipe

Figure A.1: This figure shows the steps in computing the prolongation stencil in an $[8 \times 8]$ grid. First a direct insertion from the coarse grid is performed (**??**), followed be filling the ghostlayer perpendicular to the x-axis from the neighbouring grid (**??**). Then a swipe is performed in the y-direction filling the grid points between, taking half the value from the node above, and half from the node below (**??**). Then a ghost swap is performed before doing a swap in the x-direction (**??**).

```
//Interpolation 3rd Dim
f = fSizeProd[1] + fSizeProd[2] + 2*fSizeProd[3];
fNext = f + fSizeProd[3];
fPrev = f - fSizeProd[3];

for(int l = 0; l < fTrueSize[3]; l+=2){
  for(int k = 0; k < fSize[2]; k+=2){
    for(int j = 0; j < fSize[1]; j+=2){
      fVal[f] = 0.5*(fVal[fPrev]+fVal[fNext]);
      f +=2;
      fNext +=2;
      fPrev +=2;
    }
    f   +=fSizeProd[2];
    fNext   +=fSizeProd[2];
    fPrev   +=fSizeProd[2];
  }
  f   +=fSizeProd[3];
  fNext   +=fSizeProd[3];
  fPrev   +=fSizeProd[3];
}

gSwapHalo(fine, mpiInfo, 2);

//Interpolation 2nd Dim
f = fSizeProd[1] + 2*fSizeProd[2] + fSizeProd[3];
fNext = f + fSizeProd[2];
fPrev = f - fSizeProd[2];

for(int l = 0; l < fTrueSize[3]; l++){
  for(int k = 0; k < fSize[2]; k+=2){
    for(int j = 0; j < fSize[1]; j+=2){
      fVal[f] = 0.5*(fVal[fPrev]+fVal[fNext]);
      f +=2;
      fNext +=2;
      fPrev +=2;
    }
    f   +=fSizeProd[2];
    fNext   +=fSizeProd[2];
    fPrev   +=fSizeProd[2];
  }
}

gSwapHalo(fine, mpiInfo, 1);

//Interpolation 2nd Dim
f = 2*fSizeProd[1] + fSizeProd[2] + fSizeProd[3];
fNext = f + fSizeProd[1];
fPrev = f - fSizeProd[1];
```

```
for(int l = 0; l < fTrueSize[3]; l++){
  for(int k = 0; k < fTrueSize[2]; k++){
    for(int j = 0; j < fSize[1]; j+=2){
      fVal[f] = 0.5*(fVal[fPrev]+fVal[fNext]);
      f +=2;
      fNext +=2;
      fPrev +=2;
    }
  }
  f   +=2*fSizeProd[2];
  fNext   +=2*fSizeProd[2];
  fPrev   +=2*fSizeProd[2];
}
```

Listing A.3: Codesnippet for the Z Y and X sweeps

Notes

- Restriction algorithm
  - I tried to make a simpler algorithm, that just cycled through all the values and didn't care about the ghost layers. The problem was that the fine grid needed to increment twice as fast as the coarse grid, except at the edge where it needed to increment the same as the coarse grid, due to having the same number of ghost cells.
- Prolongation
  - All the dimensional swipes repeats themself so much that it should easily be able to modified to an nDim algorithm instead of a 2D and 3D case.
- Should time everything, test performance/improvements