0.1 General idea

When an iterative solver solves a problem, it starts with an initial guess then for each cycle it improves the guess to come closer to the wanted solution. The difference between the guess and the correct solution, the residual, does not necessarily converge equally fast for different frequencies. A solver can be very efficient on reducing the local error, while it takes many cycles to reduce the errors due to distant influence. A multigrid solver attacks this problem by applying iterative methods on different discretizations of the problem, by solving on a very coarse grids the error due to distant influence will be reduced faster, while solving on a fine grid reduces the local error fast. So by solving on both fine and coarse grids the needed cycles will be reduced. To implement a multigrid algorithm we then need algorithms to solver the problem on a grid **??**, restriction **??** and prolongation **??** operators to transfer the problem between grids, as well as a method to compute the residual.

0.1.1 V-cycle

The simplest multigrid cycle is called a V-cycle, which starts at the finest grid, goes down to the coarsest grid and then goes back up to the finest grid. First the problem is smoothed on the finest level, then we compute the residual, or the rest after inserting the guess solution in the equation. The residual is then used as the source term for the next level, and we restrict it down as the source term for the next coarser level and repeat untill we reach the coarsest level. When we reach the coarsest level the problem is solved there and we obtain a correction term. The correction term is prolongated to the next finer level and added to the solution there, improving the solution, following by a new smoothing to obtain a new correction. This continue untill we reach the finest level again and a multigrid cycle is completed, see fig. 1 for a 3 level schematic.

In the following description of the steps in the MG method, we will use ϕ, ρ, d and ω to signify the solution, source, defect and correction respectively. A subscript means the grid level, where 0 si the finest level, while the superscript 0 implies an initial guess is used. Hats and tildes are also used to signify the stage the solution is in, with a hat meaning the solution is smoothed and a tilde meaning the correction from the grid below is added.

The operations necessary on a level in a multigrid cycle is given in table 1

At the coarsest level the the problem is solved directly and the correction is propageted upward.

0.1.2 Updating the Halo

All of the grids has a halo of ghosts around it, which is necessary each computational node only represents a subdomain of the whole, with the neighboring node

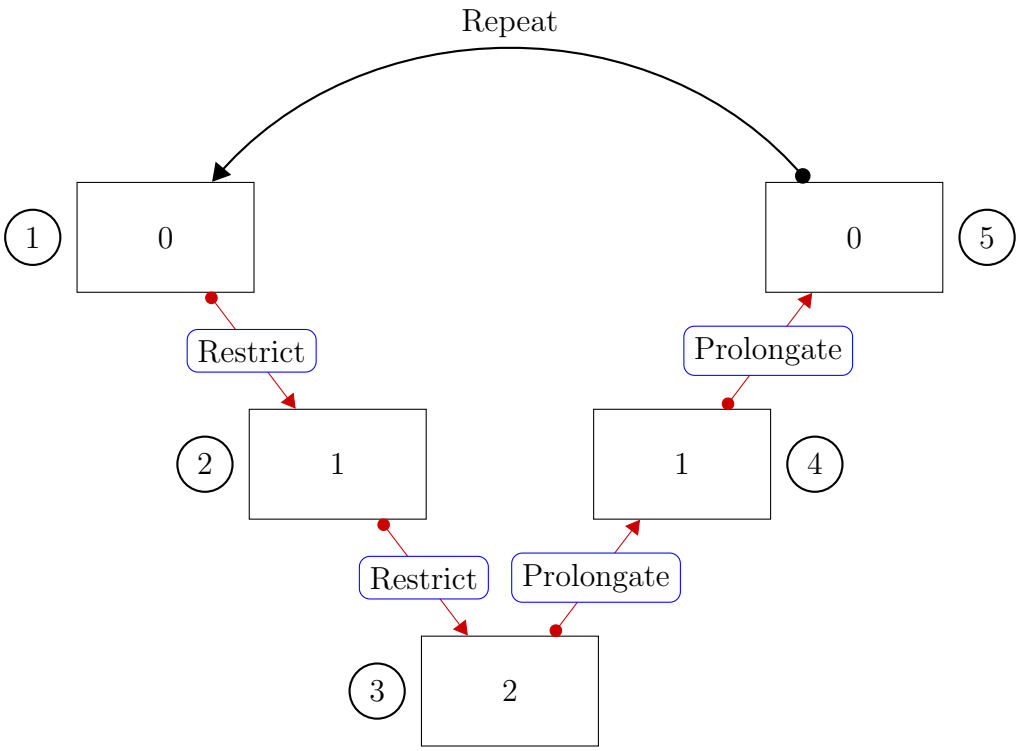


Figure 1: Schematic overview of the PIC method. In a three level MG implementation, there is 5 main steps in a cycle that needs to be considered.

1: Smooth	$\widehat{\phi}_l = \mathcal{S}(\phi_l, \rho_l)$
2: Residual	$d_l = \nabla^2 \widehat{\phi}_l - \rho_l$
3: Restrict	$\rho_{l+1} = \mathcal{R} d_l$
4: Go down, recieve correction	$\omega_l = \mathcal{I} \phi_{l+1}$
5: Add correction	$\widetilde{\phi}_l = \widehat{\phi}_l + \omega_l$
6: Smooth	$\phi_l = \mathcal{S}(\widetilde{\phi}_l, \rho_l)$
7: Interpolate correction	$\omega_{l-1} = \mathcal{I} \phi_l$

Table 1: The operations done on each level in the multigrid cycle.

being the boundary. In addition the ghost halo is used to facilitate boundary conditions on the whole domain. For some of the grid operators the ghost are not used, while some of them need updated values. All of the iterative solvers, that are used for smoothing, need updated values of the of the solution, ϕ. The prolongation and residual operators need updated values for the solution ϕ, and the restrictor need updated residual values, ρ, as long as direct insertion is not used. We also need to take into account that the smoothers outputs an updated halo for ϕ, to avoid unnecessarily communication between the processors.

0.1.3 Implementation

In general there are 4 different quantities, the source, the solution, the residual and the correction, we need to keep account of. On a grid level the residual is computed, then it is set as the source term for the next level and then it is not used more. The correction, the improvement to the finer grid, is only used when going to a finer grid. Due to this we can save some memory by letting the correction and the residual share the same memory, so both are stored in the mgRes struct. There are a regular as well as a recursive implementation of a V-cycle. The functions takes the current level, the bottom of the cycle as well as the end point of the cycle. So several different cycles can be built from the functions. A W cycle can be built a V cycle that starts at the finest level and stops at a mid level, and then a new V-cycle is started at the mid level that ends at the finest level. A full multigrid algorithm (FMG) can also be implemented by first restricting the original source term down to the coarsest level and then run a V-cycle that ends at the finest level. (Note: there will be a a general mgSolver function pointer that can be set to prebuilt cycles, so the cycle can be set from the input file)

The regular V cycle algorithm is quite straightforward, first it restricts and computes itself down to the bottom level, then it solves it directly on the bottom level. Then the correction is brought up and improved through the grid up to the top level. See **??** for an example code.

The recursive algorithm uses an algorithm more similar to the one described

in table 1. First it computes the steps necessary so the grid below has an updated source term, then it calls itself on a lower level. After recieving the correction from the lower level it is improved and sent to the level above. If the function is at the bottom level, it solves the problem directly and sends the correction up. See **??** for an example code.

0.1.4 V-cycle, code

```
void inline static mgVRecursive(int level, int bottom, int
    top, Multigrid *mgRho, Multigrid *mgPhi,
                 Multigrid *mgRes, const MpiInfo *mpiInfo){

  //Solve and return at coarsest level
  if(level == bottom){
    gInteractHalo(setSlice, mgPhi->grids[level], mpiInfo);
    mgRho->coarseSolv(mgPhi->grids[level],
        mgRho->grids[level], mgRho->nCoarseSolve, mpiInfo);
    mgRho->prolongator(mgRes->grids[level-1],
        mgPhi->grids[level], mpiInfo);
    return;
  }

  //Gathering info
  int nPreSmooth = mgRho->nPreSmooth;
  int nPostSmooth= mgRho->nPostSmooth;

  Grid *phi = mgPhi->grids[level];
  Grid *rho = mgRho->grids[level];
  Grid *res = mgRes->grids[level];

  //Boundary
  gInteractHalo(setSlice, rho, mpiInfo);
  gBnd(rho,mpiInfo);

  //Prepare to go down
  mgRho->preSmooth(phi, rho, nPreSmooth, mpiInfo);
  mgResidual(res, rho, phi, mpiInfo);
  gInteractHalo(setSlice, res, mpiInfo);
  gBnd(res, mpiInfo);

  //Go down
  mgRho->restrictor(res, mgRho->grids[level + 1]);
  mgVRecursive(level + 1, bottom, top, mgRho, mgPhi, mgRes,
      mpiInfo);

  //Prepare to go up
  gAddTo( phi, res );
  gInteractHalo(setSlice, phi,mpiInfo);
  gBnd(phi,mpiInfo);
  mgRho->postSmooth(phi, rho, nPostSmooth, mpiInfo);

  //Go up
  if(level > top){
    mgRho->prolongator(mgRes->grids[level-1], phi, mpiInfo);
  }
  return;
```

```
}
```

Listing 1: Implementation of an recursive V-cycle

```
void mgVRegular(int level, int bottom, int top, Multigrid
    *mgRho, Multigrid *mgPhi,
                    Multigrid *mgRes, const MpiInfo *mpiInfo){

  //Gathering info
  int nPreSmooth = mgRho->nPreSmooth;
  int nPostSmooth= mgRho->nPostSmooth;
  int nCoarseSolv= mgRho->nCoarseSolve;


  //Down to coarsest level
  for(int current = level; current <bottom; current ++){
    //Load grids
    Grid *phi = mgPhi->grids[current];
    Grid *rho = mgRho->grids[current];
    Grid *res = mgRes->grids[current];

    //Boundary
    gInteractHalo(setSlice, phi,mpiInfo);
    gBnd(phi,mpiInfo);

    mgRho->preSmooth(phi, rho, nPreSmooth, mpiInfo);
    mgResidual(res, rho, phi, mpiInfo);
    mgRho->restrictor(res, mgRho->grids[current + 1]);
  }

  //Solve at coarsest
  gInteractHalo(setSlice, mgRho->grids[bottom], mpiInfo);
  gBnd(mgRho->grids[bottom],mpiInfo);
  mgRho->coarseSolv(mgPhi->grids[bottom],
      mgRho->grids[bottom], nCoarseSolv, mpiInfo);
  mgRho->prolongator(mgRes->grids[bottom-1],
      mgPhi->grids[bottom], mpiInfo);

  //Up to finest
  for(int current = bottom-1; current >-1; current --){
    //Load grids
    Grid *phi = mgPhi->grids[current];
    Grid *rho = mgRho->grids[current];
    Grid *res = mgRes->grids[current];


    //Prepare to go up
    gAddTo( phi, res );
    gInteractHalo(setSlice, phi,mpiInfo);
    gBnd(phi,mpiInfo);
    mgRho->postSmooth(phi, rho, nPostSmooth, mpiInfo);
    if(level > top)
       mgRho->prolongator(mgRes->grids[current-1], phi,
       mpiInfo);
```

```
    }

    return;
}
```

Listing 2: Implementation of an recursive V-cycle

0.2 Ex: 3 level V cycle, steps necessary

(This should probably be cut.)

① Compute defect on grid 0, the finest grid:

- $\widehat{\phi}_0 = \mathcal{S}(\phi_0, \rho_0)$
- $d_0 = \nabla^2 \widehat{\phi}_0 - \rho_0$
- Restrict defect: $\rho_1 = \mathcal{R} d_0$

② Compute defect on grid 1:

- $\widehat{\phi}_1 = \mathcal{S}(\phi_1^0, \rho_1)$
- $d_1 = \nabla^2 \widehat{\phi}_1 - \rho_1$
- Restrict defect: $\rho_2 = \mathcal{R} d_1$

③ Solve Coarse Grid for correction ω

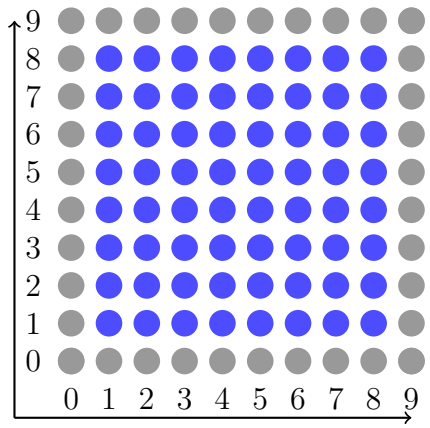
- $\phi_2 = \mathcal{S}(\phi_2^0, \rho_2)$
- Interpolate as correction:$\omega_1 = \mathcal{I}\phi_2$

④ Add correction on level 1:

- $\widetilde{\phi}_1 = \widehat{\phi}_1 + \omega_1$
- $\phi_1 = \mathcal{S}(\widetilde{\phi}_1, \rho_1)$
- Interpolate correction:$\omega_0 = \mathcal{I}\phi_1$

⑤ Compute solution.

- $\widetilde{\phi}_0 = \widehat{\phi}_0 + \omega_0$
- $\phi_0 = \mathcal{S}(\widetilde{\phi}_0, \rho_0)$



(a) Dirichlet boundary conditions

0.3 Boundary conditions

Since a simulation must necessarily have a finite extent, we need a strategy for the edges of the simulation. This model is built with support for three different boundary conditions. Periodic where the boundary is set equal to the other side of the simulation, dirichlet where the boundary has a known potential and von Neumann where the gradient of the potential is known. It is also possible to have a mixture of boundary conditions. To keep the implementation of the whole field solver modular, the boundary conditions is implemented with ghost cell so the iterative solver can work independently of the boundary conditions.

0.3.1 Periodic

With periodic boundary conditions we want the boundary on one side to be equal to the field on the other side of the plasma. In the parallelization of the simulation the domain is already divided into several smaller subdomains, where each of the subdomains needs to know the edges of the neighboring subdomains, which is stored as a halo of ghost cells around the true grid respresenting the physical subdomain. So to achieve periodic boundary conditions we just let the boundary subdomains keep the ghost layer values from the neighboring subdomains.

0.3.2 Dirichlet

Dirichlet boundary conditions