Contents

Chapter 1

Method

1.1 Poisson Solvers

In the Particle in Cell method, instead of calculating the forces directly between the particles, we first calculate the electric field from all the particles and then use that to calculate the individual force on the particles. From the particles we use a weighting scheme to obtain a charge density on a grid. Then we can use Poisson's equation eq. (1.1) to calculate the electric potential from the charge density, before computing the electric field from the potential. So we need an efficient Poisson solver.

$$\nabla^2 \Phi = -\rho \qquad \text{in} \qquad \Omega \tag{1.1}$$

The problem also need boundary conditions in which we will focus on periodic, Dirichlet and Neumann boundary conditions.

1.1.1 Spectral Methods

The spectral methods is based on Fourier transforms of the problem and solving the problem in it's spectral version, see (Shen, 1994) **(Fix citation format)** for an implementation of an spectral poisson solver. They are efficient solvers that can be less, but can be inaccurate for complex geometries.

When looking for a solution with a spectral method we first rewrite the functions as Fourier series, which for the three-dimensional Poisson equation would be

$$\nabla^2 \sum A_{j,k,l} e^{i(jx+ky+lz)} = \sum B_{j,k,l} e^{i(jx+ky+lz)} \tag{1.2}$$

From there we get a relation between the coefficients

$$A_{j,k,l} = -\frac{B_{j,k,l}}{j^2 + k^2 + l^2} \tag{1.3}$$

Then we compute the Fourier transform of the right hand side obtaining the coefficients $B_{j,k,l}$. We compute all the coefficients $A_{j,k,l}$ from the relation between the coefficients. At last we perform a inverse Fourier transform of the left hand side obtaining the solution.

(1.4)

1.1.2 Finite Element Methods

The finite element is a method to numerically solve a partial differential equations (PDE) first transforming the problem into a variational problem and then constructing a mesh and local trial functions, see Alnæs et al., 2011 **(fix norwegian letter in name)** for a more complete discussion.

To transform the PDE to a variational problem we first multiply the PDE by a test function v, then it is integrated using integration by parts on the second order terms. Then the problem is separated into two parts, the bilinear form $a(u, v)$ containing the unknown solution and the test function and the linear form $L(v)$ containing only the test function.

$$a(u, v) = L(v) \qquad v \epsilon \hat{V} \tag{1.5}$$

Next we construct discrete local function spaces of that we assume contain the trialfunctions and testfunctions. The function spaces often consists of locally defined functions that are 0 except in a close neighbourhood of a mesh point, so the resulting matrix to be solved is sparse and can be computed quickly. The matrix system is then solved by a suiting linear algebra algorithm, before the solution is put together.

1.2 Multigrid

The multi grid, MG, method used to solve the Poisson equation and obtain the electric field is a widely used and highly efficient solver for elliptic equations, having a theoretical scaling of $\mathcal{O}(N)$ (Press et al., 1988), where N is the grid points. Here I will go through the main theory and algorithm behind the method, as explained in more detail in (Press et al., 1988; Trottenberg et al., 2000) as well as go through some of possible algorithms to parallelize the method.

We want to solve a linear elliptic problem,

$$\mathcal{L}u = f \tag{1.6}$$

where $\mathcal{L}$ is a linear operator, u is the solution and f is a source term. In our specific case the operator is given by the laplacian, the source term is given by the charge density and we solve for the electric potential.

We discretize the equation onto a grid of size q.

$$\mathcal{L}_q u_q = f_q \tag{1.7}$$

Let the error, v_q be the difference between the exact solution and an approximate solution to the difference equation (1.7), $v_q = u_q - \tilde{u}_q$. Then we define the residual as what is left after using the approximate solution in the equation.

$$d_q = \mathcal{L}_q \tilde{u}_q - f_q \tag{1.8}$$

Since $\mathcal{L}$ is a linear operator the error satisfies the following relation

$$\mathcal{L}_q v_q = \mathcal{L}(u_q - \tilde{u}_q) + (f_q - f_q) \tag{1.9}$$
$$\mathcal{L}_q v_q = -d_q \tag{1.10}$$

In the multigrid methods instead of solving the equation directly here, we set up a system nested coarser square grids, $\mathfrak{T}_1 \subset \mathfrak{T}_2 \subset \cdots \subset \mathfrak{T}_\ell$, where 1 is the coarsest grid and ℓ is the finest. Then the main thought behind the methods is that instead of solving the problem directly on the initial grid, we use restriction, $\mathcal{R}$, and interpolation, $\mathcal{P}$, operators to change the problem between the different grid levels and solve them on there. (Fix previous sentence) Due to the fewer grid points the problem is faster to solver on the coarser grid levels than on the fine grid.

If we then apply a restriction operator on the residual we go down a level on the grids and the opposite for the interpolation operator.

$$\mathcal{R} d_q = d_{q-1} \qquad \text{and} \qquad \mathcal{P} d_q = d_{q+1} \tag{1.11}$$

1.2.1 Algorithm

A sequential algorithm for a two grid V shaped algorithm, where the coarse grid and fine grid has respectively $q = 1, 2$.

- Initial approximation. $\tilde{u}$
- for $i <$ nCycles:
  - Presmooth: $\hat{u}_2 = S_{pre}(u)_2$

- Calculate defect: $d_2 = f_2 - \mathcal{L}\hat{u}_2$
- Restrict defect: $d_1 = Rd_2$
- Initial guess: $\tilde{u}_1 = 0$
- Solve (G-S RB): $L_1\tilde{u}_1 = d_1$
- Interpolate: $\tilde{u}_2 = I\tilde{u}_1$
- Add correction: $u_2^{new} = \hat{u}_2 + \tilde{u}_2$

1.2.2 Smoothing: Gauss-Seidel

Relaxation methods, such as Gauss-Seidel, work by looking for the setting up the equation as a diffusion equation, and then solve for the equilibrium solution.

So suppose we want to solve the elliptic equation

$$\mathcal{L}u = \rho \tag{1.12}$$

Then we set it up as a diffusion equation

$$\frac{\partial u}{\partial t} = \mathcal{L}u - \rho \tag{1.13}$$

By starting with an initial guess for what u could be the equation will relax into the equilibrium solution $\mathcal{L}u = \rho$. By using a Forward-Time-Centered-Space scheme to discretize, along with the largest stable timestep $\Delta t = \Delta^2/2^d$ (Double check the factor.), we arrive at Jacobi's method, which is an averaging of the neighbors in addition to a contribution from the source term. By using the already updated values for in the calculation of the u^{new} we arrive at the method called Gauss-Seidel which for two dimensions is the following

$$u_{i,j}^{n+1} = \frac{1}{4}\left(u_{i+1,j}^{n} + u_{i-1,j}^{n+1} + u_{i,j+1}^{n} + u_{i,j-1}^{n+1}\right) - \frac{\Delta^2\rho_{i,j}}{4} \tag{1.14}$$

To achieve a vectorization of the calculation of $u_{i,j}^{n+1}$ we will use Red and Black ordering, first calculating the odd nodes and then the even nodes, so the method has the available information.

1.3 Parallelization

For the parallelization of an algorithm there is two obvious issues that need to be addressed to ensure that the algorithm retains a high degree of parallelization; communication overhead and load imbalance (Hackbusch and Trottenberg, 1982). Communication overhead means the time the computational nodes spend

communicating with each other, if that is longer than the actual time spent computing the speed of the algorithm will suffer, and load imbalance appears if some nodes need to do more work than others causing some nodes to stand idle.

Here we will focus on multigrid of a 3D cubic (to be expanded to rectangular cubes) grid, where each grid level has half the number of grid points. We will use grid partitioning to divide the domain, GS-RB (Gauss-Seidel Red-Black) as both a smoother and a coarse grid solver.

We need to investigate how the different steps: interpolation, restriction, smoothing and the coarse grid solver, in a MG method will handle parallelization.

1.3.1 Grid Partition

There are several well explored options for how a multigrid method can be parallized, for example Domain Decomposition (**arraras˙domain˙2015**), Algebraic Multigrid (Stüben, 2001), see Chow et al. (2006) for a survey of different techniques. Here we will focus on Geometric Multigrid (GMG) with grid partitioning used for the parallelization, as described in the books Trottenberg et al., 2000; Hackbusch and Trottenberg, 1982.

With grid partitioning we divide the grid $\mathcal{T}$ into geometric subgrids, then we can let each processes handle one subgrid each, as we will see it can be useful when using the G-S RB smoothing to let the subgrids overlap 1 layer deep. since it on the edges of the subgrid it will need the adjacent node values.

1.3.2 Distributed and accumulated data

During the parallel execution of the code there can be useful to keep track of the different data structures and what needs to be accumulated over all the computational nodes and what only needs to be distributed on the individual computational nodes.

- u solution (Φ)
- w temporary correction
- d defect
- f source term (ρ)
- $\mathcal{L}$ differential operator
- $\mathcal{I}$ interpolation operator
- $\mathcal{R}$ restriction operator
- $\mathbf{u}$ Bold means accumulated vector

- $\tilde{\mathbf{u}}$ is the temporary smoothed solution
- Accumulated vectors: $\mathbf{u}_q$, $\hat{\mathbf{u}}_q$, $\tilde{\mathbf{u}}_q$, $\hat{\mathbf{w}}_q$ $\mathbf{w}_{q-1}$, $\mathbf{I}$,$\mathbf{R}$
- Distributed vectors: f_q, d_q, d_{q-1}

Algorithm: P is the number of processes

- If (q == 1): Solve: $\sum_{s=1}^{P} \mathcal{L}_{s,1}\mathbf{u}_1 = \sum_{s=1}^{P} f_{s,1}$
- else:
  - Presmooth: $\hat{\mathbf{u}}_q = \mathcal{S}_{pre}\mathbf{u_q}$
  - Compute defect: $d_q = f_q - \mathcal{L}_q\hat{\mathbf{u}}_q$
  - Restrict defect: $d_{q-1} = \mathcal{R}d_q$
  - Initial guess: $\mathbf{w}_{q-1} = 0$
  - Solve defect system: $\mathbf{w}_{q-1} = PMG(\mathbf{w}_{q-1}, d_{q-1})$
  - Interpolate correction: $\mathbf{w}_q = \mathcal{R}\mathbf{w}_{q-1}$
  - Add correction: $\tilde{\mathbf{u}}_q = \hat{\mathbf{u}}_q + \mathbf{w}_q$
  - Post-smooth: $\mathbf{u}_q = \mathcal{S}_{post}$

1.3.3 Smoothing

We have earlier divided the grid into subgrids, with overlap, as described in subsection 1.3.1 and given each processor responsibility for a subgrid. Then do a a GS-RB method we start with an approximation of $u_{i,j}^n$. Then we will obtain the next iteration by the following formula

$$u_{i,j}^{n+1} = \frac{1}{4}\left(u_{i+1,j}^n + u_{i-1,j}^{n+1} + u_{i,j+1}^n + u_{i,j-1}^{n+1}\right) - \frac{\Delta^2\rho_{i,j}}{4} \tag{1.15}$$

We can see that the for the inner subgrid we will have no problems since we have all the surrounding grid points. On the edges we will need the adjacent grid points that are kept in the other processors. To avoid the algorithm from asking neighboring subgrids for adjacent grid points each time it reaches a edge we instead update the entire neighboring column at the start. So we will have a 1-row overlap between the subgrids, that need to be updated for each iteration.

1.3.4 Restriction

For the transfer down a grid level, to the coarser grid we will use a half weighting stencil. In two dimensions it will be the following

$$\mathcal{R} = \frac{1}{8}\begin{bmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{1.16}$$

With the overlap of the subgrids we will have the necessary information to perform the restriction without needing communication between the processors (Hackbusch and Trottenberg, 1982).

1.3.5 Interpolation

For the interpolation we will use bilinear interpolation:

$$\mathcal{I} = \frac{1}{4}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{1.17}$$

Since the interpolation is always done after GS-RB iterations the the outer part overlapped part of the grid updated, and we can have all the necessary information.

1.3.6 Scaling

Volume-Boundary effect

While a sequential MG algorithm has a theoretical scaling of $\mathcal{O}(N)$ (Press et al., 1988), where N is the number of grid points, an implementation will have a lower scaling efficiency due to interprocessor communication. We want a parallel algorithm that attains a high speedup with more added processors P, compared to sequential 1 processor algorithm. Let $T(P)$ be the computational time needed for solving the problem on P processors. Then we define the speedup $S(P)$ and the parallel efficiency $E(P)$ as

$$S(P) = \frac{T(1)}{T(P)} \qquad E(P) = \frac{S(P)}{P} \tag{1.18}$$

A perfect parallel algorithm would the computational time would scale inversely with the number of processors, $T(P) \propto 1/P$ leading to $E(P) = 1$. Due to the necessary interprocessor communication that is generally not achievable.

	Cycle	Sequential	Parallel
MG	V	$\mathcal{O}(N)$	$\mathcal{O}(\log N)$
	W	$\mathcal{O}(N)$	$\mathcal{O}\left(\sqrt{N}\right)$
FMG	V	$\mathcal{O}(N)$	$\mathcal{O}\left(\log^2 N\right)$
	W	$\mathcal{O}(N)$	$\mathcal{O}\left(\sqrt{N}\log N\right)$

Table 1.1: The parallel complexities of sequential and parallel multigrid cycles

The computational time of the algorithm is also important, if the algorithm is very slow but has good parallel efficiency it is often worse than a fast algorithm with a worse parallel efficiency.

The parallel efficiency of an algorithm is governed by the ratio between the time of communication and computation, T_{comm}/T_{comp}. If there is no need for communication, like on 1 processor, the algorithm is perfectly parallel efficient. In our case the whole grid is diveded into several subgrids, which is assigned to different processors. In many cases the time used for computation is roughly scaling with the interior grid points, while the communication time is scaling with the boundaries of the subgrids. If a local solution method is used on a local problem it is only the grid points at the boundary that needs the information from grid points on the other processors. Since the edges has lower dimensionality than the inner grid points. So when the problem is is increasing the time for computation grows faster than the time for communication, and a parallel algorithm often has a higher parallel efficiency on a larger problem. This is called the Boundary-Volume effect.

Parallel complexity

The complexities, as in needed computational work, of sequential and parallel MG cycles is calculated in Hackbusch and Trottenberg, 1982 and is shown in table 1.1. In the table we can see that in the parallel case there is a substantial increase in the complexity in the case of W cycles compared to V cycles. In the sequential case the change in complexity when going to a W cycle is not dependent on the problem size, but it is in the parallel case.

1.4 Grid Structs and Partitioning

1.5 Data structures

The fields and quantities in our PiC model is discretized on a threedimensional grid and comes to use several places in the method. In the multigrid calculation

we also have a use for several grids of varying spatial coarseness. So it will be useful for us to organize the data so we have the grid stored as an independent structure available for the program, while the multigrid part uses an extended version where it also has access to the different coarser subgrids. There will also be several types of grid, all with the same specifications, but storing different quantities, so we will use a grid template that all the grids of the same size shares, while we will have a seperate struct containing the quantities in the grid and the specifications.

1.6 Domain partitioning

As earlier discussed, in 1.3.1 the physical domain covered by our model will be divided onto several processors that each take care of a subdomain. The subdomains are dependent on each other and we need some communication between them, which we solve by letting each subdomain also store the edge of the neighboring subdomain. Depending on the boundary conditions it could also be useful to store an extra set of values on the outer domain boundary as well, which will be called ghost points, N_G. The extra grid points due the the overlap between the subdomains we will call overlap points, N_O. Let us for simplicity sake consider a regular domain, with equal extent in all dimensions, with N grid points per dimension, d and consider how many grid values we need to store as a singular domain and the grid values needed when it is divided amongst several processors, a 2 dimensional case is depicted in fig. 1.1.

1.6.1 Singular domain

In the case where the whole domain is worked on by one process we need N^d to store the values on the grid representing the physical problem, in addition we see that we also need to store values for the ghost points along the domain boundary. Given that we have one layer of ghost points on all the boundaries, and there is 2 boundaries per dimension, the total number of ghost points is given by $N_G = 2dN$. Since there is only 1 domain we don't need to account for any overlap between subdomains and the total grid points we need to store is:

$$N_{Tot} = N^d + N_G + N_O = N^d + 2dN^{d-1} \tag{1.19}$$

For the 2 dimensional case, in fig. 1.1, that adds up to $N_{Tot} = 8^2 + 2 \times 4 \times 8 = 128$.

1.6.2 Several subdomains

In the case where we introduce several subdomain, in addition to storing the grid values and the ghost points we also need to store an overlap between the subdomains. If we take our whole domain Ω and divide in up into several small domains Ω_S, the smaller domains only takes a subsection of the grid points. For simplicity case, and equal load on processors, we let the subdomains as well be regular, with the whole domain being a multiple of the subdomains. Our whole domain has N grid points in each direction, if we then divide that domain into $\#\Omega$ domains, then each of those subdomains will have $N_S = N^d/\#\Omega$ grid points. Each of those subdomains will also need values representing the ghost points and overlap from the neighboring nodes. A boundary of a subdomain will either have overlap points, or ghost points, not both at the same time so for each boundary we need to 1 layer, N_S^{d-1}. Each subdomain will have 2 boundaries per dimension since we have regular subdomains. The total number of grid points needed per subdomain is then

$$N_{Tot,S} = N_S^d + (N_G + N_O) = N_S^d + 2dN_S^{d-1} \tag{1.20}$$

while the total number of grid points is

$$N_{Tot} = \#\Omega N_{Tot,S} \tag{1.21}$$

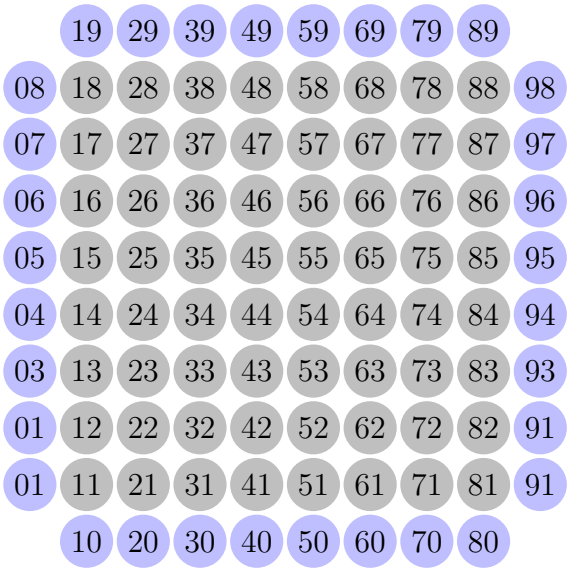
For the 2 dimensional case discussed earlier we need $N_{Tot,S} = 4^2 + 2 \times 2 \times 4^1 = 32$.

Since the effect of the subdomain boundaries increase the coarser the grid is we should not let the coarsest multigrid level be to small. We also don't need the spatial extent of the grid to be equal on all sides, but it was done here to keep the computations simple.

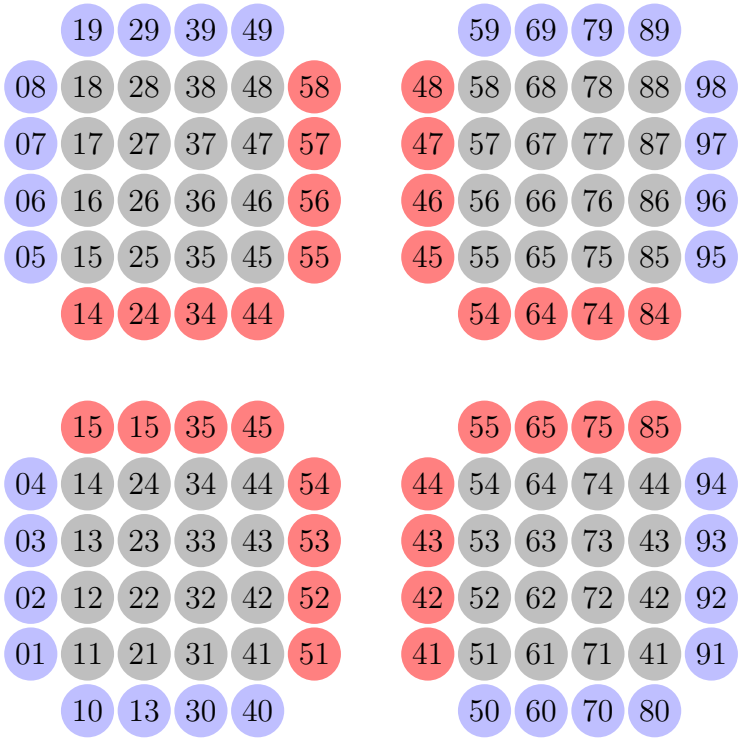
1.7 Implementation of MG parts

1.8 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 (1.22), for 1D, 2D and 3D examples.



(a) The grid points needed for an 8×8 domain.



(b) The 8×8 grid divided into 4 subdomains

Figure 1.1: Each circle in the figures represents 1 grid point, and the first number is the column while the second is the row. The grey colour represents physical space the computational node works on, the blue color is the outer grid points for boundary conditions and the red colour is the overlapping grid points.

$$\begin{aligned}
\mathcal{R}_{1D} &= \frac{1}{4}\begin{bmatrix}1 & 2 & 1\end{bmatrix} \\
\mathcal{R}_{2D} &= \frac{1}{8}\begin{bmatrix}0 & 1 & 0\\ 1 & 4 & 1\\ 0 & 1 & 0\end{bmatrix} \\
\mathcal{R}_{3D} &= \frac{1}{12}\left(\begin{bmatrix}0 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 0\end{bmatrix}, \begin{bmatrix}0 & 1 & 0\\ 1 & 6 & 1\\ 0 & 1 & 0\end{bmatrix}, \begin{bmatrix}0 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 0\end{bmatrix}\right)
\end{aligned} \tag{1.22}$$

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 1.1: 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 1.2: The foor loop doing the calculations

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

1.9 Prolongation

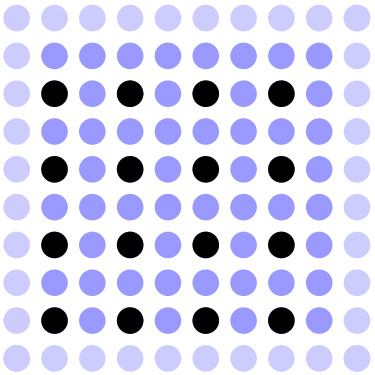
Along with the restriction operator described in the previous section, we also need prolongation operator to go from a coarse grid to a finer grid. Here we will use bilinear interpolation, for two dimensions and trilinear interpolation for 3 dimensions. In bilinear interpolation seperate linear interpolation 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 (1.23)

$$\begin{aligned}\mathcal{I}_{2\mathrm{D}} &= \frac{1}{4}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix}\\ \mathcal{I}_{3\mathrm{D}} &= \frac{1}{8}\left(\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix}, \begin{bmatrix}2 & 4 & 2\\ 4 & 8 & 4\\ 2 & 4 & 2\end{bmatrix}, \begin{bmatrix}2 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix}\right)\end{aligned} \tag{1.23}$$

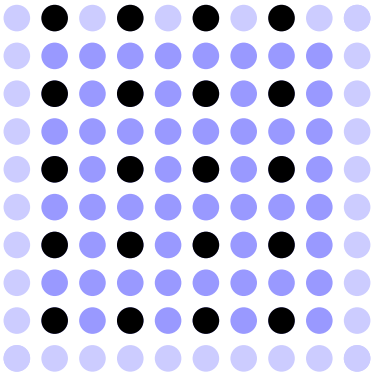
The algorithm implemented for the interpolation is based on the method, described in **??** has the following steps, which is also shown for a 2D case in 1.2.

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

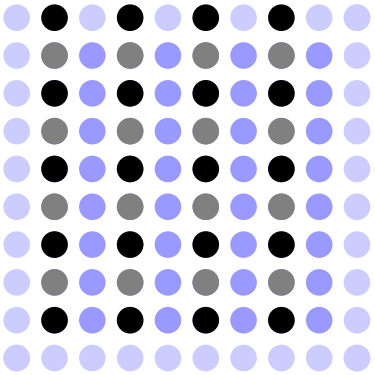
The interpolation should always first be done on the highest dimension, because the grid values are stored further apart along the highest axis in the memory, and the each succesive interpolation needs to apply to more grid points. (Note to self: should test)



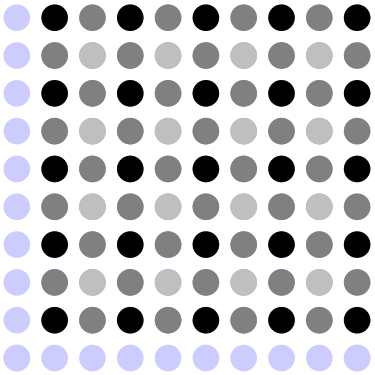
(a) Direct insertion



(b) Swapping X-Dim ghosts



(c) Y-swipe



(d) Swapping Y-ghostlayer and X-swipe

Figure 1.2: 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 (1.2a), followed be filling the ghostlayer perpendicular to the x-axis from the neighbouring grid (1.2b). 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 (1.2c). Then a ghost swap is performed before doing a swap in the x-direction (1.2d).

The direct insertion is done similar to the previously described in the restriction section, **??**.

```
//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 1.3: 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.
  - When filling ghost layer in a particular direction, we already know that only one side has values. So transferring both sides of the ghost layer is unnecessary.
- Should time everything, test performance/improvements

1.10 Jacobian and Gauss-Seidel RB

The main iterative ODE solver, in this version of the multigrid program, is a Gauss-Seidel Red-Black, in addition a Jacobian solver was developed as a stepping stone and testing purposes. It is a modification of the Jacobian method, where the updated values are used where available, which lead to it converging twice as fast **NumReci**

Our problem is given by $\nabla^2\phi = -\rho$, one way to think of the jacobian method is as a diffusion problem, and with the equilibrium solution as our wanted solution. If we then discretize the diffusion problem by a Forward-Time-Centralized-Space scheme, we arrive at the Jacobian method, which is shown explicitly below for 1 dimension.

$$\frac{\partial\phi}{\partial t} = \nabla^2\phi + \rho \tag{1.24}$$

The subscript j indicates the spatial coordinate, and the superscript n is the 'temporal' component.

$$\frac{\phi_j^{n+1} - \phi_j^{n+1}}{\Delta t} = \frac{\phi_{j+1}^n - 2\phi_j^n + \phi_{j-1}^n}{\Delta x^2} + \rho_j \tag{1.25}$$

This is numerically stable if $\Delta t/\Delta x^2 \leq 1/2$, so using the timestep $\Delta t = \Delta x^2/2$ we get

$$\phi_j^{n+1} = \phi_j^n + \frac{1}{2}\left(\phi_{j+1}^n - 2\phi_j^n + \phi_{j-1}^n\right) + \frac{\Delta x^2}{2}\rho_j \tag{1.26}$$

Then we arrive at the Jacobian method

$$\phi_j^{n+1} = \frac{1}{2}\left(\phi_{j+1}^n + \phi_{j-1}^n + \Delta x^2\rho_j\right) \tag{1.27}$$

The Gauss-Seidel method uses updated values, where available, and is given by

$$\phi_j^{n+1} = \frac{1}{2}\left(\phi_{j+1}^n + \phi_{j-1}^{n+1} + \Delta x^2\rho_j\right) \tag{1.28}$$

Following the same procedure we get the Gauss-Seidel method for for 2 and 3 dimensions.

$$\phi_{j,k}^{n+1} = \frac{1}{4}\left(\phi_{j+1,k}^n + \phi_{j-1,k}^{n+1} + \phi_{j,k+1}^n + \phi_{j,k-1}^{n+1} + \Delta x^2\rho_{j,k}\right) \tag{1.29}$$

$$\phi_{j,k,l}^{n+1} = \frac{1}{8}\left(\phi_{j+1,k,l}^n + \phi_{j-1,k,l}^{n+1} + \phi_{j,k+1,l}^n + \phi_{j,k-1,l}^{n+1} + \phi_{j,k,l+1}^n + \phi_{j,k,l-1}^{n+1} + \Delta x^2\rho_{j,k,l}\right) \tag{1.30}$$

Here we have implemented a different version of the Gauss-Seidel algorithm called Red and Black ordering, which has conseptual similarities to the leapfrog algorithm, where usually position and velocity is computed at t and $t + (\delta t)/2$. Every other grid point is labeled a red point, and the remaining is black. When updating a red node only black nodes are used, and when updating black nodes only red nodes are used. Then a whole cycle consists of two halfsteps which calculates the red and black nodes seperately.

- For all red points:

$$\phi_{j,k,l}^{n+1/2} = \frac{1}{8}\left(\phi_{j+1,k,l}^{n} + \phi_{j-1,k,l}^{n} + \phi_{j,k+1,l}^{n} + \phi_{j,k-1,l}^{n} + \phi_{j,k,l+1}^{n} + \phi_{j,k,l-1}^{n} + \Delta x^2 \rho_{j,k,l}\right)$$

- For all black points:

$$\phi_{j,k,l}^{n+1} = \frac{1}{8}\left(\phi_{j+1,k,l}^{n+1/2} + \phi_{j-1,k,l}^{n+1/2} + \phi_{j,k+1,l}^{n+1/2} + \phi_{j,k-1,l}^{n+1/2} + \phi_{j,k,l+1}^{n+1/2} + \phi_{j,k,l-1}^{n+1/2} + \Delta x^2 \rho_{j,k,l}\right)$$

1.10.1 Implementation

Jacobi's method

The implementation of the jacobian algorithm is straightforward, but it has the downside of slow convergence and bad smoothing properties, in addition to needing an additional created. When ϕ_i^{n+1} is computed we need access to the previous value ϕ_{i-1}^n, and more values in higher dimensions, so either the previous values need to stored seperately, or ϕ_i^{n+1} can be computed on a new grid and then copied over after completing the cycle. In this implementation we computed on a temporary grid and then copied over, since it was mostly for debugging purposes and efficiency was not a concern.

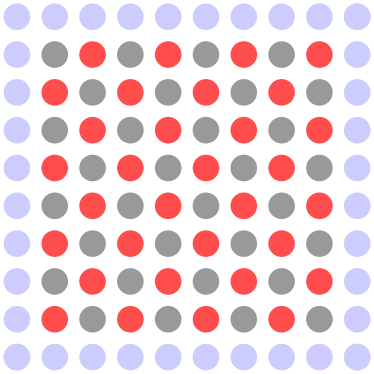
The computation is done by starting at index $g = 0$, computing the surrounding grid indexes, $gj, gjj, ...$, where gj is the next grid point along the x-axis, and gjj is the previous value. Then the entire grid is looped trough over, incrementing both g and the surrounding grid indexes $gj, gjj, ...$ The computation on the ghost layers will be incorrect but those will be overwritten when swapping halos. See 1.10.2 for an example code in 2D.

Gauss-Seidel Red and Black

In the implementation of Gauss-Seidel algorithm we use a clever ordering of the computations, called Red and Black ordering, both to increase the smoothing properties of the algorithm as well as avoiding creating a temporary grid to store ϕ^{n+1} in. Every grid point where the indexes sum up to an even number is labeled a red point and the odd index groupings are labeled black points, see fig. 1.3a. Then each red point is directly surrounded by only black points and vica versa.

A cycle is then divided into 2 halfcycles, where each halfcycle computes ϕ^{n+1} for the red and black points respectively.

1. for(int c = 0; c¡nCycles; c++)
    - Cycle through red points and compute ϕ^{n+1}
    - Swap Halo
    - Cycle through black points and compute ϕ^{n+1}



(a) Red and Black ordering

- Swap Halo

For the 2 dimensional case the cycling is done first for the odd rows and even rows seperately, due to the similarity between all the red points in the odd rows, and between the red points in the even rows. Then the cycling could be generalized into a static inline function used for all the cycling. See 1.10.3 for the 2D implementation.

For the 3 dimensional case there is now did two different takes on the problem, one where the iteration through the grid is streamlined, but needing several loops through the grid taking care of a subgroup of the grid points each loop. The other algorithm uses one loop through the grid, with different conditions on the edges to make it go through the correct grid points in each line.

When the loops are streamlined the edges the loop go through the entire grid, but when it reaches and edge in it either needs to add or subtract 1 to the iterator index. In we want to do a red pass, computing all the red values, the grid is cycled through increasing by 2 each time. Then it will access up the indexes, $36, 38, 40, 42, \cdots$. In the second row we want it to use the index 43, instead of 42, so we need to increase it by 1 when it reaches the edge. When it reaches the end of the second line, we want it to increase from 47 to 48, so then we need to subtract 1. In the next layer we need to shift the behaviour on the edges to the opposite. See

.	.	.	.	.	.
48	49	49	50	51	52
42	43	44	45	46	47
36	37	38	39	40	41

In the other implementation I tried something similar to the 2D implementation, where it does several loops through the grid, computing an easier subgroup of the red nodes each time, so the iteratior index can increase by just to each time. So for the red points it computes the odd and even layers and rows seperately.

- Compute Odd layers, odd rows
- Compute Odd layers, even rows
- Compute Even Layers, odd rows
- Compute Even layers, even rows

1.10.2 Jacobian code

```
for(int c = 0; c < nCycles; c++){
  // Index of neighboring nodes
  int gj = sizeProd[1];
  int gjj= -sizeProd[1];
  int gk = sizeProd[2];
  int gkk= -sizeProd[2];

  for(long int g = 0; g < sizeProd[rank]; g++){
    tempVal[g] = 0.25*( phiVal[gj] + phiVal[gjj] +
                phiVal[gk] + phiVal[gkk] + rhoVal[g]);

    gj++;
    gjj++;
    gk++;
    gkk++;
  }

  for(int q = 0; q < sizeProd[rank]; q++) phiVal[q] =
     tempVal[q];
  for(int d = 1; d < rank; d++) gSwapHalo(phi, mpiInfo,
     d);
}
```

Listing 1.4: Code snippet 2D jacobian

1.10.3 GS-RB 2D

```
    for(int c = 0; c < nCycles;c++){

      //Increments
      int kEdgeInc = nGhostLayers[2] + nGhostLayers[rank +
          2] + sizeProd[2];

      /*************************
       *  Red Pass
       *************************/
      //Odd numbered rows
      g = nGhostLayers[1] + sizeProd[2];
      loopRedBlack2D(rhoVal, phiVal, sizeProd, trueSize,
          kEdgeInc, g, gj, gjj, gk, gkk);

      //Even numbered columns
      g = nGhostLayers[1] + 1 + 2*sizeProd[2];
      loopRedBlack2D(rhoVal, phiVal, sizeProd, trueSize,
          kEdgeInc, g, gj, gjj, gk, gkk);

      for(int d = 1; d < rank; d++) gSwapHalo(phi, mpiInfo,
          d);

      /*********************************
       *  Black pass
       *********************************/
      //Odd numbered rows
      g = nGhostLayers[1] + 1 + sizeProd[2];
      loopRedBlack2D(rhoVal, phiVal, sizeProd, trueSize,
          kEdgeInc, g, gj, gjj, gk, gkk);

      //Even numbered columns
      g = nGhostLayers[1] + 2*sizeProd[2];
      loopRedBlack2D(rhoVal, phiVal, sizeProd, trueSize,
          kEdgeInc, g, gj, gjj, gk, gkk);


      for(int d = 1; d < rank; d++) gSwapHalo(phi, mpiInfo,
          d);
    }

    return;
  }
```

Listing 1.5: Main loop

```
  gj = g + sizeProd[1];
  gjj= g - sizeProd[1];
```

```
      gk = g + sizeProd[2];
      gkk= g - sizeProd[2];

      for(int k = 1; k < trueSize[2]; k +=2){
        for(int j = 1; j < trueSize[1]; j += 2){
          phiVal[g] = 0.25*(  phiVal[gj] + phiVal[gjj] +
                      phiVal[gk] + phiVal[gkk] + rhoVal[g]);
          g +=2;
          gj  +=2;
          gjj +=2;
          gk  +=2;
          gkk +=2;
        }
        g +=kEdgeInc;
        gj  +=kEdgeInc;
        gjj +=kEdgeInc;
        gk  +=kEdgeInc;
        gkk +=kEdgeInc;
      }
```

Listing 1.6: Loop through grid

1.11 GS-RB 3D if tests

```
/*********************
 *  Red Pass
 *********************/
g = sizeProd[3]*nGhostLayers[3];
for(int l = 0; l < trueSize[3];l++){
  for(int k = 0; k < size[2]; k++){
    for(int j = 0; j < size[1]; j+=2){
      phiVal[g] = 0.125*( phiVal[g+gj] + phiVal[g-gj] +
                phiVal[g+gk] + phiVal[g-gk] +
                phiVal[g+gl] + phiVal[g-gl] + rhoVal[g]);
      g +=2;
    }
    if(l%2){
      if(k%2) g+=1; else g-=1;
    } else {
      if(k%2) g-=1; else g+=1;
    }

  }
  if(l%2) g-=1; else g+=1;
}

for(int d = 1; d < rank; d++) gSwapHalo(phi, mpiInfo, d);
```

Listing 1.7: GS-RB with if-tests

1.12 GS-RB 3D without if tests

```
/************************
 *  Red Pass
 ************************/
//Odd layers - Odd Rows
g = nGhostLayers[1]*sizeProd[1] +
    nGhostLayers[2]*sizeProd[2] +
    nGhostLayers[3]*sizeProd[3];
loopRedBlack3D(rhoVal, phiVal, sizeProd, trueSize,
    kEdgeInc, lEdgeInc,
         g, gj, gjj, gk, gkk, gl, gll);

//Odd layers - Even Rows
g = (nGhostLayers[1]+1)*sizeProd[1] +
    (nGhostLayers[2]+1)*sizeProd[2] +
    nGhostLayers[3]*sizeProd[3];
loopRedBlack3D(rhoVal, phiVal, sizeProd, trueSize,
    kEdgeInc, lEdgeInc,
         g, gj, gjj, gk, gkk, gl, gll);

//Even layers - Odd Rows
g = (nGhostLayers[1])*sizeProd[1] +
    (nGhostLayers[2])*sizeProd[2] +
    (nGhostLayers[3]+1)*sizeProd[3];
loopRedBlack3D(rhoVal, phiVal, sizeProd, trueSize,
    kEdgeInc, lEdgeInc,
         g, gj, gjj, gk, gkk, gl, gll);

//Even layers - Even Rows
g = (nGhostLayers[1] + 1)*sizeProd[1] +
    (nGhostLayers[2]+1)*sizeProd[2] +
    (nGhostLayers[3]+1)*sizeProd[3];
loopRedBlack3D(rhoVal, phiVal, sizeProd, trueSize,
    kEdgeInc, lEdgeInc,
         g, gj, gjj, gk, gkk, gl, gll);

for(int d = 1; d < rank; d++) gSwapHalo(phi, mpiInfo, d);
```

Listing 1.8: main routine

```
  inline static void loopRedBlack3D(double *rhoVal,double
     *phiVal,long int *sizeProd, int *trueSize, int
     kEdgeInc, int lEdgeInc,
        long int g, long int gj, long int gjj, long int gk,
           long int gkk, long int gl, long int gll){

  gj = g + sizeProd[1];
  gjj= g - sizeProd[1];
```

```
    gk = g + sizeProd[2];
    gkk= g - sizeProd[2];
    gl = g + sizeProd[3];
    gll= g - sizeProd[3];

    for(int l = 0; l<trueSize[3]; l+=2){
      for(int k = 0; k < trueSize[2]; k+=2){
        for(int j = 0; j < trueSize[1]; j+=2){
          // msg(STATUS, "g=%d", g);
          phiVal[g] = 0.125*(phiVal[gj] + phiVal[gjj] +
                  phiVal[gk] + phiVal[gkk] +
                  phiVal[gl] + phiVal[gll] + rhoVal[g]);
          g +=2;
          gj  +=2;
          gjj +=2;
          gk  +=2;
          gkk +=2;
          gl  +=2;
          gll +=2;
        }
      g +=kEdgeInc;
      gj  +=kEdgeInc;
      gjj +=kEdgeInc;
      gk  +=kEdgeInc;
      gkk +=kEdgeInc;
      gl  +=kEdgeInc;
      gll +=kEdgeInc;
      }
    g +=lEdgeInc;
    gj  +=lEdgeInc;
    gjj +=lEdgeInc;
    gk  +=lEdgeInc;
    gkk +=lEdgeInc;
    gl  +=lEdgeInc;
    gll +=lEdgeInc;
    }

    return;
  }
```

Listing 1.9: loop routine

1.13 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 section 1.10, restriction section 1.8 and prolongation section 1.9 operators to transfer the problem between grids, as well as a method to compute the residual.

1.13.1 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. 1.4 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 1.2

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

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

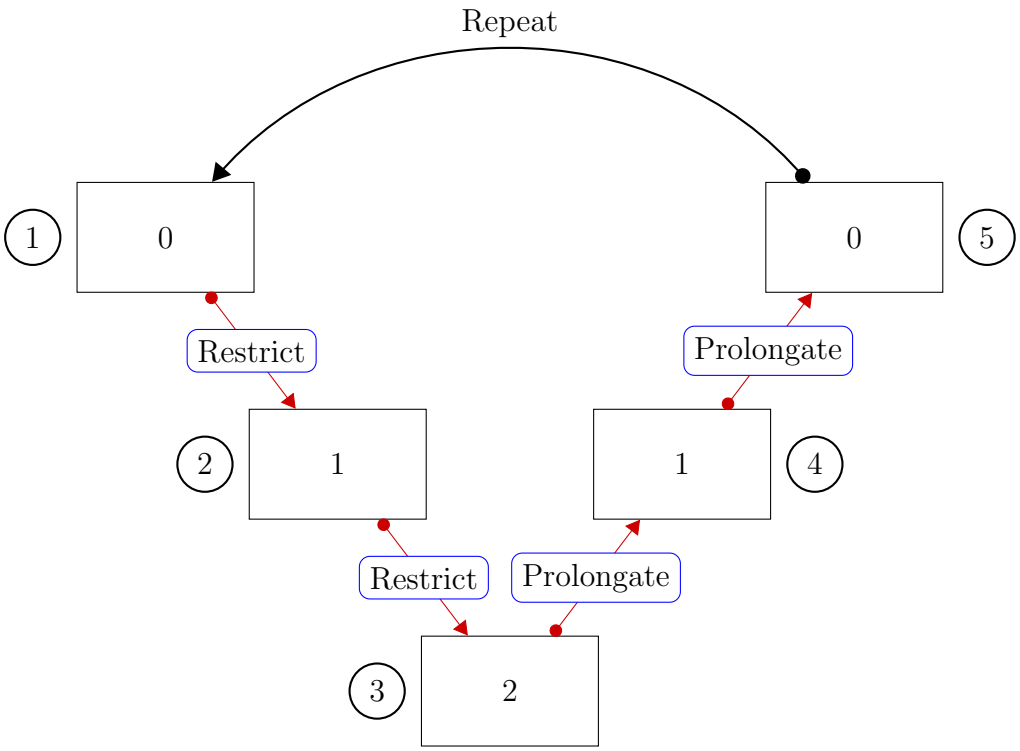


Figure 1.4: Schematic overview of the PIC method. In a three level MG implementation, there is 5 main steps in a cycle that needs to be considered.

1.13.2 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 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.

1.13.3 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 1.2. 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.

1.13.4 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 1.10: 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 1.11: Implementation of an recursive V-cycle

1.14 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

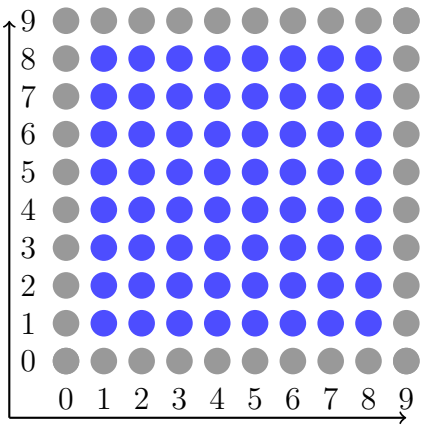
- $\phi_2 = \mathcal{S}(\phi_2^0, \rho_2)$
- Interpolate as correction:$\omega_1 = \mathcal{I}\phi_2$

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

1.15 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.

1.15.1 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.

1.15.2 Dirichlet

Dirichlet boundary conditions

Chapter 2

Verification

2.1 Electron plasma oscillations in unmagnetized plasma

2.1.1 Physical mechanism

Here we will consider an electron fluid under local thermal equilibrium, LTE, experiencing small perturbations from an equilibrium state of an homogeneous electron density, n_0, and pressure, p_0, and vanishing flow, $\mathbf{u}_0 = 0$, and electric field, $\mathbf{E}_0 = 0$. See Goldston and Rutherford (1995) for a more thorough overview.

The small perturbation of the electron density will cause an electric field electric field working to restore the equilibrium. When the electrons reach the equilibrium position they have a kinetic energy and will overshoot causing a new perturbation away from the equilibrium.

The fluid is governed by the following equations:

$$\frac{\partial n_e}{\partial t} + \nabla \cdot (n_e \mathbf{u}_e) = 0 \tag{2.1a}$$

$$m_e n_e \left(\frac{\partial}{\partial t} + \mathbf{u}_e \cdot \nabla \right) \mathbf{u}_e = e n_e \nabla \phi - \nabla p_e \tag{2.1b}$$

$$\left(\frac{\partial}{\partial t} + \mathbf{u}_e \cdot \nabla \right) p_e + \frac{5}{3} p_e \nabla \cdot \mathbf{u}_e = 0 \tag{2.1c}$$

$$\epsilon_0 \nabla^2 \phi = e \left(n_e - n_0 \right) \tag{2.1d}$$

Assuming the small perturbation to the equilibrium.

$$\text{Perturbation} \rightarrow \begin{cases} n_e = n_0 + \tilde{n}_e \\ p_e = p_0 + \tilde{p}_e \\ \mathbf{u}_e = \tilde{\mathbf{u}}_e \\ \phi = \tilde{\phi} \end{cases}$$

Inserting the perturbation and linearizing the equations we get:

$$\frac{\partial \tilde{n}_e}{\partial t} + \nabla \cdot (n_0 \tilde{\mathbf{u}}_e) = 0 \tag{2.2a}$$

$$m_e \frac{\partial \tilde{\mathbf{u}}_e}{\partial t} = e \nabla \tilde{\phi} - \frac{\nabla \tilde{p}_e}{n_0} \tag{2.2b}$$

$$\frac{\partial \tilde{p}}{\partial t} + \frac{5}{3} p_0 \nabla \cdot \tilde{\mathbf{u}}_e = 0 \tag{2.2c}$$

$$\epsilon_0 \nabla^2 \tilde{\phi} = e \tilde{n}_e \tag{2.2d}$$

Then we combine the continuity and energy equations, eq. (2.2a) and eq. (2.2c).

$$\frac{\partial}{\partial t} \left(\frac{\tilde{p}_e}{p_0} + \frac{5}{3} \frac{\tilde{n}_e}{n_0} \right) = 0 \tag{2.3}$$

The perturbed pressure and density is proportional, $\nabla \tilde{p}_e = (5p_0/3n_0)\, \nabla \tilde{n}_e$. Assuming plane wave solutions along the x-axis, so the differential operators become $\nabla \to ik$ and $\frac{\partial}{\partial t} \to -i\omega$, we can solve for the dispersion relation.

$$\epsilon(\omega, k) = 1 + \frac{5}{3} \lambda_{se}^2 k^2 - \frac{\omega^2}{\omega_{pe}^2} \tag{2.4}$$

Here we have subsituted in the electron debye length $\lambda_{se} = (\epsilon_0 p_0)/(e^2 n_0^2)$, and the electron plasma frequency $\omega_{pe} = e^2 n_0$

2.1.2 Test Case

Since the physics behind the electron plasma wave is quite simple, we want to verify that our code works by letting it reproduce the waves.

- Homogenous plasma
- No fields
- Equilibrium? (Lattice structure?) First test this
- No thermal velocity / (Not necessary?)
-

Appendix A

Optional methods

A.1 MG-Methods

A.1.1 FMG

- 'Easy' to implement.
- Theoretically scales $\mathcal{O}(N)$ (**Press1987**), in reality
- FMG and W cycles are usually avoided in massive parallel computers (Chow et al., 2006), as they visit the coarsest grid often, due to:
  - At the coarsest level the computation is fast, communication usually the bottleneck
  - Coarsest grid may couple all the domain, needs global communication
- Options to solve the coarse matrix (Chow et al., 2006)
  - Direct solver: Sequential, if small problem may be done on each processor to avoid communication
  - Iterative method, Gauss-Seidel. Can be parallelized
- Will use G-S with R-B ordering, has good parallel properties

A.2 Other methods worth considering

- MUMPS (MUltifrontal Massive Parallel Solver), tried and compared in **Kacem2012** slower than MG.
  - Solves $Au = \rho$ for a sparse matrix.

Appendix B

Multigrid Libraries

Efficient computation of the poisson equation, or other elliptic equations, is a common problem with many applications, and there exists several predeveloped and optimized libraries to help solve it. These include Parallel Particle Mesh (PPM) (**Sbalzarini2006**), Hypre (**Falgout02hypre:a**), Muelu (??), METIS (**METIS**) and PETCs (**Gropp2001**) amongst others. There is also PiC libraries that can be used PICARD and VORPAL to mention two.

If we want to have an efficient integration of a multigrid library into our PiC model we need to consider how easy it is to use with our scalar and field structures. To have an effiecient program we need to avoid having the program convert data between our structures and the library structures. Since our PiC implementation uses the same datastructures for the scalar fields in several other parts, than the solution to the poisson equation, we could have an efficiency problem in the interface between our program and the library.

We could also consider that only part of the multigrid algorithm uses building blocks from libraries. The algorithm is now using the conceptually, and programatically easy, GS-RB as smoothers, but if we implement compatibility with a library we could easily use several other types of smoothers which could improve the convergence of the algorithm

B.1 Libraries

B.2 PPM - Parallel Particle Mesh

Parallel Particle Mesh is a library designed for particle based approaches to physical problems, written in Fortran. As a part of the library it includes a structured geometric multigrid solver which follows a similar algorithm to the algorithm we have implemented in our project implemented in both 2 and 3 dimensions. For the 3 dimensional case the laplacian is discretized with a 7-point stencil, then it uses a RB-SOR (Red and Black Succesive Over-Relaxation), which equals GS-

RB with the relaxation parameter ω set as 1, as a smoother. The full-weighting scheme is used for restriction and trilinear interpolation for the prolongation, both are described in (**Trottenberg**). It has implementations for both V and W multigrid cycles. To divide up the domains between the computational nodes it uses the METIS library. The efficiency of the parallel multigrid implementation was tested

B.3 Hypre

Hypre is a library developed for solving sparse linear systems on massive parallel computers. It has support for c and Fortran. Amongst the algorithms included is both structured multigrid as well as element-based algebraic multigrid. The multigrid algorithms scales well on up to 100000 cores, for a detailed overview see Baker et al 2012. (bibtex files started to argue, will fix).

B.4 MueLo - Algebraic Multigrid Solver

MueLo is an algebraic multigrid solver, and is a part of the TRILINOS project and has the advantage that it works in conjunction with the other libraries there. It is written as an object oriented solver in cpp. For a investigation into the scaling properties see Lin et. al. 2014.

B.5 METIS - Graph Partitioning Library

METIS is a library that is used for graph partitioning, and could have been used in our program to partition the grids. The partitionings it produces has been shown to be 10% to 50% faster than the partionionings produces by spectral partitioning algorithms (**Karypis**). It is mostly used for irregular graphs, and we are not sure if it could be easily made to work with the datastructures used throughout the program.

B.6 PETSc - Scientific Toolkit

The PETSc is an extensive toolkit for scientific calculation that is used by a multitude of different numerical applications, including FEniCS. It has a native multigrid option, DMDA, where the grid can be constructed as a cartesian grid. In addition there is large amount of inbuilt smoothers that can be used.