Contents

Chapter 1

Theoretical Background

1.1 Plasma

If you, the reader, is already familiar with plasma physics this section could serve as as a quick reminder about important basic plasma theory. Or it could hopefully serve as a too short and shallow introduction, to the uninitatied reader, of the knowledge needed to make sense of the numerical experiments in this thesis. For a more thorough introduction the books *Plasma Physics* (Fitzpatrick, 2014), *Introduction to Plasma Physics* (Goldston and Rutherford, 1995), *Waves and Oscillations in Plasmas* (Pécseli, 2012) and the classic *Introduction to Plasma Physics and Controlled Fusion* (Chen, 1984) can be consulted.

- What
- Where
- Why

1.1.1 Plasma Parameters

Let us first consider an idealized plasma, with approximately equal number of ions and electrons. Each of the species has mass m_s, where the subscript signifies specie, and respectively charge $-e$, $+e$. Then we let the plasma be in a quasi-neutral state so the number density, n, is apprimately equal, $n_i \approx n_e = n$. Now let's introduce the concept of kinetic temperature T_s, i.e. the random motion part of the kinetic energy.

$$T_s \equiv \frac{1}{3} m_s \left\langle v_s^2 \right\rangle$$

With the kinetic temperature we define the thermal speed, v_{ts}, as

$$v_{ts} \equiv \sqrt{2T_s}/m_s$$

here we should note that the electron thermal speed is usually much larger than the ion thermal speed due to the mass proportion between them.

Time scales in plasma is usually related to the plasma frequency, ω_{pe}, of the electron as the fastest gyrating particle.

$$\omega_{pe} = \frac{ne^2}{\epsilon_0 m_e}$$

The reciprocal of the plasma frequency, the plasma period, $\tau_p \equiv 1/\omega_{pe}$ is often helpful as well.

Then we define the Debye length λ_D as the length a typical particle travels in a plasma period, ignoring the ion contribution.

$$\lambda_D \equiv v_t \tau = \sqrt{\frac{\epsilon_0 T}{ne^2}}$$

For a plasma description to be useful the system we consider must have a typical length scale, L, and time scale, τ, larger than the Debye length and plasma period respectively.

$$\frac{\lambda_D}{L} \ll 1 \qquad ; \qquad \frac{\tau_p}{\tau} \ll 1$$

1.2 Single Particle Motion

To better understand the collective motion of plasma it is useful to consider the motions of the single particles that the plasma consists of. By at first treating only one particle we can ignore the electromagnetic influence from the other particles which greatly simplifies the situation. The Lorentz term is often the dominant force in a plasma due to it's relative strength compared to gravity and other forces.

$$\mathbf{F} = q\left(\mathbf{E} + \mathbf{v} \times \mathbf{B}\right) \tag{1.1}$$

To simplify matters we will only consider particles in static electromagnetic field, as that is often a valid approximation on the time and spatial scales particle gyration operates on.

1.2.1 Gyration

Let us consider a situation with a single particle in a static and isotropic external magnetic field $\mathbf{B}$ and the particle has an initial velocity $\mathbf{v}$. The equation of motion for the particle will is then

$$m\frac{\partial v}{\partial t} = q\mathbf{v} \times \mathbf{B} \tag{1.2}$$

The velocity parallel to the magnetic field is unaffected by the field. So we will seperate the velocity into a parallel and perpendicular velocity with respect to the magnetic field, $\mathbf{v} = \mathbf{v}_\parallel + \mathbf{v}_\perp$. Then we remove the uninteresting parallel part and we do a temporal derivative on the equation of motion.

$$\frac{\partial^2 \mathbf{v}_\perp}{\partial t^2} = \frac{q}{m}\frac{\partial \mathbf{v}_\perp}{\partial t} \times \mathbf{B} \tag{1.3}$$

Then we insert eq. (1.2) into the equation and use the vector relation $a \times b \times c = b(a \cdot b) - c(a \cdot b)$.

$$\frac{\partial^2 \mathbf{v}_\perp}{\partial t^2} + \left(\frac{qB}{m}\right)^2 \mathbf{v}_\perp = 0 \tag{1.4}$$

This differential equation corresponds to a gyration around the magnetic field lines with the plasma frequency, $\omega_p = \frac{qB}{m}$, as the frequency. From here on we will use the term $\omega_\perp$ to refer to the gyration part of the velocity described here.

1.2.2 E-cross-B Drift

The E-cross-B drift happens when a particle is moving with static and isotropic electric and magnetic fields. The equation of motion is then

$$m\frac{\partial v}{\partial t} = q(\mathbf{E} + \mathbf{v} \times \mathbf{B}) \tag{1.5}$$

As in the previous section we decompose the velocity into a parallel and perpendicular velocity. We also assume there exists a constant drift $\mathbf{v}_D$, somewhat unfounded at this stage, so that we can seperate the perpendicular motion into the drift and the gyration. The velocity is then given by $\mathbf{v} = \mathbf{v}_\parallel + \boldsymbol{\omega}_\perp + \mathbf{v}_D$, and we insert this into the equation of motion.

$$m\frac{\partial}{\partial t}\left(\mathbf{v}_\parallel + \boldsymbol{\omega}_\perp + \mathbf{v}_D\right) = q\left(\mathbf{E} + \left(\mathbf{v}_\parallel + \boldsymbol{\omega}_\perp + \mathbf{v}_D\right) \times \mathbf{B}_0\right) \tag{1.6}$$

The parallel velocity is unaffected by the magnetic field and has uninteresting solutions only involving the electric field. From section 1.2.1 we know that the gyration part is given by

$$m\frac{\partial \omega_\perp}{\partial t} = q\omega_\perp \times \mathbf{B} \tag{1.7}$$

When we take out these two parts we have left the drift velocity

$$\frac{\partial \mathbf{v}_D}{\partial t} = \frac{q}{m}\left(\mathbf{E}_\perp + \mathbf{v}_D \times \mathbf{B}\right) \tag{1.8}$$

Then we use the assumption that the drift velocity is constant, cross the equation with $\mathbf{B}$ and perform some algebra to arrive at

$$\mathbf{v}_D = \frac{\mathbf{E} \times \mathbf{B}}{B^2} \tag{1.9}$$

As we can see the E-cross-B drift is independent of particle charge and mass, which means both the ions and electrons will be drifting at the same velocity subject to the drift which will be perpendicular to electric and magnetic fields.

1.3 Fluid Description

1.3.1 Fluid Equations

The generalized fluid equations is given in eqs. (1.10) to (1.12), where the three equations describe the conservation of mass, momentum and energy respectively. We refer you to the earlier mentioned *Plasma Physics* for the rather involved derivation to obtain them. The symbols n_s, $\mathbf{u}_s$, p_s, $\mathbf{p}_s$, $\mathbf{F}_s$ and $\mathcal{W}_s$ stands respectively for the specie specific; number density, averaged velocity, scalar pressure, pressure tensor, force and energy.

$$\left(\frac{\partial n_s}{\partial t} + \mathbf{u}_s \cdot \nabla\right) n_s + n_s \nabla \cdot \mathbf{u}_s = 0 \tag{1.10}$$

$$m_s n_s \left(\frac{\partial}{\partial t} + \mathbf{u}_s \cdot \nabla\right) \mathbf{u}_s + \nabla \cdot p_s - e_s n_s \left(\mathbf{E} + \mathbf{e}_s \times \mathbf{B}\right) = \mathbf{F}_s \tag{1.11}$$

$$\frac{3}{2}\left(\frac{\partial}{\partial t} + \mathbf{u_s} \cdot \nabla\right) p_s + \frac{3}{2} p_s \nabla \cdot u_s + \mathbf{p}_s : \nabla \mathbf{u_s} + = \mathcal{W}_s \tag{1.12}$$

1.3.2 Plasma States

It is usual to classify plasma into different states according to which approximations and simplifications it is valid to apply to close the fluid equations in section 1.3.1.

Local Thermodynamic Equilibrium

A plasma is said to be in a *local thermodynamic equalibrium* (LTE) if the phase-space distribution is locally maxwellian.

$$\mathcal{F}_m = \frac{n}{(2\pi)^{3/2} v_t^3} \exp\left\{ -\frac{(v-u)^2}{2v_t^2} \right\} \tag{1.13}$$

Since the viscosity tensor, π, and the heat flux tensor, $\mathbf{q}$ contains odd integrals over the distribution, *see Plasma Physics*, they dissappear. This simplifies the fluid equations.

Cold Plasma

Isothermal

1.3.3 Plasma Kinetic Theory

Another useful way to make analyze plasma is by the use of fluid approximations. The plasma is then characterized by local parameters describing particle density, kinetic temperature, flow velocity and so on. The time evolution is then governed by the fluid equation, but unfortunately the resulting equations are generally less tractable than the usual hydrodynamical equations.

Let $\mathcal{F}_s$ be the exact phase-space density of a particle species, it contains all the positions, velocities for all the particles at all times. By integrating over all velocities and multiplying with the charge for all species we obtain the charge density, ρ_c.

$$\rho_c = \sum_s e_s \int \mathcal{F}_s(\mathbf{r}, \mathbf{v}, t)\, \mathrm{d}^3\mathbf{v}$$

Likewise we find he current density, $\mathbf{j}$.

$$\mathbf{j} = \sum_s e_s \int \mathbf{v}\mathcal{F}_s(\mathbf{r}, \mathbf{v}, t)\, \mathrm{d}^3\mathbf{v}$$

Then in principle all plasma interaction could be derived from phase-space conservation, combined with the Lorentz force as the only relevant force,

$$\frac{\partial \mathcal{F}}{\partial t} + \mathbf{v} \cdot \nabla \mathcal{F}_s + \frac{e_s}{m_s} (\mathbf{E} + v \times \mathbf{B}) \cdot \nabla_v \mathcal{F}_s = 0 \tag{1.14}$$

where ∇_v is the velocity grad-operator. Unfortunately this expression, combined with Maxwells equations, is in general not tractable. To simplify matters we can average over an ensemble to obtain the average distribution function $\bar{\mathcal{F}}_s$,

$$\bar{\mathcal{F}}_s \equiv \langle \mathcal{F}_s \rangle_{ensemble}$$

.

The averaged distribution is then used in eq. (1.14) to make it more tractable. The electric, $\mathbf{E}$, and magnetic fields, $\mathbf{B}$, is statistically dependent of the distribution function and this leads to correlations when using the averaged distribution. We will not go into those here, so see *Plasma Physics*(Fitzpatrick, 2014) for a treatment of it.

1.4 Langmuir Oscillations

Here we will consider an one specie plasma fluid consisting of electrons under local thermal equilibrium, LTE. The electron density, n_0 and pressure p_0 is initally homogenous. The fluid has a vanishing flow, $\mathbf{u}_0 = 0$, and the electric field $\mathbf{E}_0 = 0$. See one of the books mentioned in section 1.1 for a more thorough overview.

A small perturbation of the electron density will cause the electric field to try to restore the equilibrium. When the electrons reach the equilibrium position they will have a kinetic energy and will overshoot.This will cause a new perturbation away from the equilibrium. This phenomen is called Langmuir oscillations.

Under the LTE conditions the fluid equations simplify to

$$\frac{\partial n_e}{\partial t} + \nabla \cdot (n_e \mathbf{u}_e) = 0 \tag{1.15a}$$

$$m_e n_e \left(\frac{\partial}{\partial t} + \mathbf{u}_e \cdot \nabla \right) \mathbf{u}_e = e n_e \nabla \phi - \nabla p_e \tag{1.15b}$$

$$\left(\frac{\partial}{\partial t} + \mathbf{u}_e \cdot \nabla \right) p_e + \frac{5}{3} p_e \nabla \cdot \mathbf{u}_e = 0 \tag{1.15c}$$

Since this set of equations have more unknowns than equations so we need additional information to close the set. Here we can use the Poisson equation to close it.

$$\epsilon_0 \nabla^2 \phi = e \left(n_e - n_0 \right) \tag{1.16}$$

Now we let a small perturbation happen to the equilibrium.

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

Since the pertubation is small, we can say that any part that contains second order terms of perturbation of a quantity will be much smaller than the value of the quantity, $q \ll \tilde{q}\tilde{q}$. So even though we may miss some processes by doing this we can drop the second order perturbation terms. This process is called linearization.

So inserting the perturbation and linearizing the equations we get:

$$\frac{\partial \tilde{n}_e}{\partial t} + \nabla \cdot (n_0 \tilde{\mathbf{u}}_e) = 0 \tag{1.17a}$$

$$m_e \frac{\partial \tilde{\mathbf{u}}_e}{\partial t} = e\nabla\tilde{\phi} - \frac{\nabla \tilde{p}_e}{n_0} \tag{1.17b}$$

$$\frac{\partial \tilde{p}}{\partial t} + \frac{5}{3} p_0 \nabla \cdot \tilde{\mathbf{u}}_e = 0 \tag{1.17c}$$

$$\epsilon_0 \nabla^2 \tilde{\phi} = e\tilde{n}_e \tag{1.17d}$$

Then we combine the continuity and energy equations, eq. (1.17a) and eq. (1.17c).

$$\frac{\partial}{\partial t}\left(\frac{\tilde{p}_e}{p_0} + \frac{5}{3}\frac{\tilde{n}_e}{n_0}\right) = 0 \tag{1.18}$$

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, the differential operators become $\nabla \to ik$ and $\frac{\partial}{\partial t} \to -i\omega$, we can solve for the dispersion relation.

$$\epsilon(\omega, k) = 1 + \frac{5}{3}\lambda_{se}^2 k^2 - \frac{\omega^2}{\omega_{pe}^2} \tag{1.19}$$

Here we have subsituted in the electron debye length λ_D and the plasma frequency ω_p.

(NOTE TO SELF: NEED TO GET TO expression for charge density time evolution)

1.5 Numerical Simulations

As there are several theoretical branches within the field of plasma physics, magnetohydrodynamics, kinetic theory (Note to self: Need better overview, and citations), that are suited to investigate plasma physics at different scales and different phenonema, there are also different approaches to conduct numerical plasma studies. Plasma simulation codes can be classified along the extent they are using a kinetic kinetic or fluid description of the plasma. Kinetic codes include Vlasov simulations (cite), Fokker-Planck simulations (Cite) and particle codes like the Particle-in-Cell code, that the development of, was a large part of this master thesis. Plasma fluid simulations are called MHD and are based on magnetohydrodynamical theory, (mention some). In the fluid description some of the detailed physics is averaged out and this causes MHD codes to be unsuited to study results depending on some small scale phenomena. Their advantage is

that due to the reduced detail they can simulate on a much larger scale. Kinetic simulations generally have more detail and capture more physics (rewrite), and as a tradeof they are restricted to simulate over a physical domain due to limited computation power and memory storage. Since the relevant timescales vary vastly between ions and electrons a multitude of hybrid codes has also been developed. (Search for multitude of hybrid codes and ref). These types of codes can e.g. treat some of the species as fluids and some as particles capturing the wanted phenomena. Particle based codes can also be combined with molecular dynamics code, if the algorithm is unsuited in a regime.

Chapter 2

Implementation of MG

2.1 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.

2.1.1 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 **??** 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 2.1

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 2.1: 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.

2.1.2 Updating the Halo

All of the subgrids has a halo of ghostslayers around it, which is used to simplify boundary conditions and subdomain communication. Each computational node represents a subdomain of the whole, with the neighboring node being the boundary. So between two subdomains each subdomain updates the boundary according to the neigbouring subdomain. In addition the 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. (NOTE TO SELF: belong in paralellization part) We also need to take into account that the smoothers outputs an updated halo for ϕ, to avoid unnecessarily communication between the processors.

2.1.3 Implementation

In general there are 4 different quantities, the source, the solution, the residual and the correction, we need to keep track of. On each 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 2.1. 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.

2.2 Restriction

In our implementation 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.

2.3 Prolongation

The algorithm implemented for the interpolation is based on the method, described in Press et al., 1988, has the following steps, which is also shown for a 2D case in 2.1.

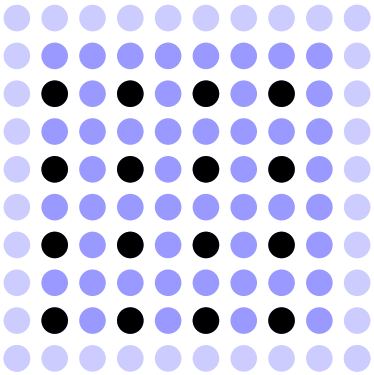
1. Direct insertion: Coarse→ Fine
2. Interpolation on highest Dimension: $f(x) = \frac{f(x+h)+f(x-h)}{2h}$
3. Fill needed ghosts.
4. Interpolation on next highest Dimension

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)

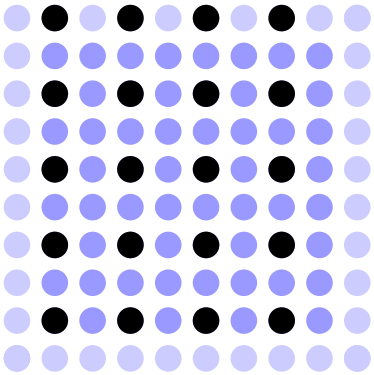
2.3.1 Implementation

Jacobi's method

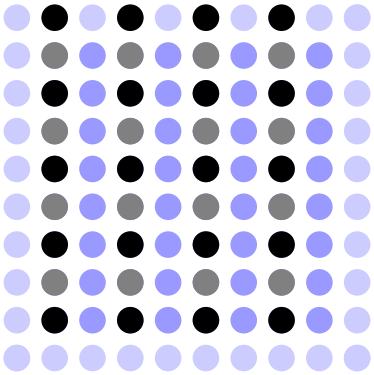
The implementation of the jacobian algorithm is straightforward, but it has the downside of slow convergence and bad smoothing properties, in addition to needing an additional grid values. When ϕ_i^{n+1} is computed we need access to the previous value ϕ_{i-1}^n, and more values in higher dimensions, 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



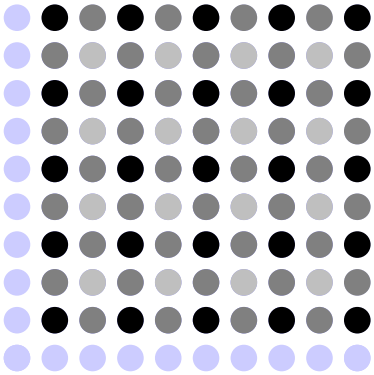
(a) Direct insertion



(b) Swapping X-Dim ghosts



(c) Y-swipe



(d) Swapping Y-ghostlayer and X-swipe

Figure 2.1: 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 (2.1a), followed be filling the ghostlayer perpendicular to the x-axis from the neighbouring grid (2.1b). 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 (2.1c). Then a ghost swap is performed before doing a swap in the x-direction (2.1d).

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 B.1.1 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. 2.2a. Then each red point is directly surrounded by only black points and vica versa.

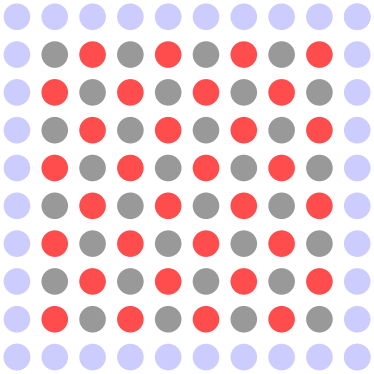
A cycle is then divided into 2 halfcycles, where each halfcycle computes ϕ^{n+1} for the red and black points respectively.

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}
    - 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 **??** 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



(a) Red and Black ordering

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

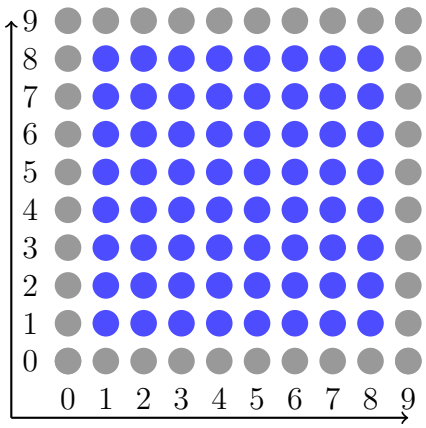
.	.	.	.	.	.
48	49	49	50	51	52
42	43	44	45	46	47
36	37	38	39	40	41

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

2.4 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.



(a) Dirichlet boundary conditions

2.4.1 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.

2.4.2 Dirichlet

Dirichlet boundary conditions

Chapter 3

Verification and Performance

In this chapter we will go through different methods we used to verify the multigrid solver, as well as scaling measurements. Modular parts of the solver is tested with unittests where feasible. In addition the whole solver is tested with both analytically solvable test cases and randomly generated fields.

(NOTE TO SELF: When ($dx \neq dy \neq dz$ the error magnitude is different for each direction, should be smaller with more grid levels)

3.0.3 Error Quantification

In order to evaluate solutions we will use two different measurements, the error, $\bar{E}$, based on the average deviation from a correct solution and the residual, $\bar{r}$ based on the average residual. Both methods is independent of the problemsize so differently sized problems can be compared.

$$\bar{E} = \frac{1}{N^d \bar{\phi}} \left(\sum_i \hat{\phi}_i - \phi_i \right) \tag{3.1}$$

$$\bar{r} = \frac{1}{N^d} \left(\sum_i \nabla^2 \hat{\phi}_i + \rho_i \right) \tag{3.2}$$

3.0.4 Analytical Solutions

We use a few different constructed charge density fields, which is analytically solvable, to test the performance and correctness of the solver. All the simulations here are ran on a grid of the size $128, 64, 64$ divided into $1, 2, 2$ subdomains. It uses 5 cycles when presmoothing, solving on the coarsest grid and postsmoothing, the MG solver is instructed to run for 100 MG V-cycles with 2 grid levels.

Figure 3.1: This a x, y-plane from the grids cut along $z_l = 32$, from the sinusoidal test case described in section 3.0.4. The left plot shows the charge distribution, the center plot shows the numerical solution of the potential and the plot to the right depicts the residual.

Sinusoidal function

A sinusoidal source term, ρ can be useful to test the solver since it can be constructed to have very simple derivatives and integrals. Here we use a sinusoidal function that has two positive tops and two negative tops over the total domain. We want the sinus function to go over 1 period over the domain, so we normalize the argument by dividing the grid point value, x_j, y_k, z_l, by the domain length in the direction, L_x, L_y, L_z.

$$\rho(x_j, y_j, z_l) = \sin\left(x_j \frac{2\pi}{L_x}\right) \sin\left(y_k \frac{2\pi}{L_y}\right) \tag{3.3}$$

A potential that fits with this is:

$$\phi(x, y, z) = -\left(\frac{2\pi}{L_x}\right)^2 \left(\frac{2\pi}{L_y}\right)^2 \sin\left(x_j \frac{2\pi}{L_x}\right) \sin\left(y_k \frac{2\pi}{L_y}\right) \tag{3.4}$$

The fig. 3.1 shows the results from running the MG-solver on the test sinusoidal test case described here. As can be expected the potential mirrors the charge distribution, except with an opposite sign and a larger amplitude. A decently large grid was simulated and the mean residual was found to be: $\bar{r} \approx 0.0312$.

Heaviside Function

The solver is also tested with a charge distribution governed by a Heaviside function. This is also suited to testing since the charge distribution is then constant planes, and we expect second order polynomial when integrating them. In the test case there are two planes with the value -1 and two planes with 1. In fig. 3.2 the test case, as well as the solution and residual is shown, and we can see the polynomials in the solution. The mean residual $\bar{r}$ was 0.00677.

$$\rho_(x_j, y_k, z_l) = \begin{cases} 1 & y_j \epsilon (0, 32), (64, 96) \\ -1 & y_j \epsilon (33, 65), (97, 127) \end{cases} \tag{3.5}$$

Figure 3.2: As earlier this is a x, y-plane cut along $x_k = 32$, of the grid. The plots show the charge distribution, numerical solution and the solution, from left to right. This is a test case constructed with Heaviside functions. In the solution of the potential the expected second degree polynomial can be seen.

Figure 3.3: As earlier this is a x, y-plane cut along $x_k = 32$, of the grid. The plots show the charge distribution, numerical solution and the solution, from left to right.

3.0.5 Random Charge distribution

To hopefully avoid some problems, that could appear due to the earlier test cases being to constructed being to orderly, a test with a randomized charge distribution is also included. The fig. 3.3 shows the charge distribution, numerical potential and the residual. The mean residual was found to be $\bar{r} \approx 0.00388$.

3.0.6 Convergence of Residual

3.0.7 Different Domain divisions

3.1 Scaling

In this section we investigate the performance of the solver and different scaling measurements. We are interested in both how well the solver performs on a larger number of processors, as well as the perfomance impact of the different parameters in the solver.

We want to obtain a better understanding of how the field resolution can be scaled up without hampering the perfomance of the particle-in-cell simulation to much.

3.1.1 Perfomance Optimizer

A multigrid solver has several parameters that needs to be set correctly for a an optimal performance (Find Source for!!!). These parameters are dependent on the problem size, as well as the computing architecture. Instead of attempting to estimate them beforehand we have included an external script that runs the program with different MG-solver settings on the wanted domainsize and tries to optimize them. The parameters it tries to change is the number of grid levels and the cycles to run for presmoothing, postsmoothing and the coarse solver. It should be worth it to spend some computing power, finding close to optimal settings, prior to running a full scale simulation since the solver needs to run each

time step. The performance optimizer naively runs the solver for a predetermined mesh of settings.

1. Smarter testing algorithm 'if better when increase continue increasing, else stop'

3.1.2 Langmuir Oscillation Test Case

As a test of the validity of our PiC program, we can use the langmuir wave oscillation. This test is inspired by a case set up in Birdsall and Langdon (2004), and modified to fit with our normalization and discretizations.

.1 Notation

While the notation is described in the main text at its first instance, we have also included this small note on the notation used to make it easier to look up. Notation that are only used in locally in smaller sections are not included here.

In the PinC project we have decided to try to keep different indexes tied different objects to help avoid confusion and increase readability. Since the i index is reserved for incrementing particles, the spatial x, y, z-indexes are j, k, l instead of the more usual i, j, k. So to make the transition between this document and the code easier we have also used the j, k, l indexes to denote the spatial area. This is the convention used by Birdsall and Langdon, (cite plasma physics via simulation).

Subscripts are usually used to denote spatial index, and a superscripts are usually reserved for temporal cases. So $\Phi^n_{j,k,l}$ means the potential at the timestep n and position j, k, l. When plasma theory is involved the subscript can also signify the particle species.

Φ	Electric Potential
ρ	Charge Density

Appendix A

Unittests

A.1 Unittests

Unittests are small tests that is used to check that the single pieces of the code work as they should. This serves a dual purpose in developing a software project. When a part of the code is developed it serves as a framework to create a standardized test of the piece of code that can easily be repeated. It also helps when developing the higher level algorithms, in that the unittests ensures that the problem lies in the higher level algorithm and not in the lower level pieces it uses. When implementing wider changes, for example datastructures, the unittests can help making sure that the changes are not causing any unintended bugs. For information of how to use the unittests see the documentation, **documentation**

A.1.1 Prolongation and Restriction

The prolongation and restriction operators with the earlier proposed stencils will average out the grid points when applied. So the idea here is to set up a system with a constant charge density, $\rho(\mathbf{r}) = C$, and then apply a restriction. After performing the restriction we can check that the grid points values are preserved. Then we can do the same with the prolongation. While this does not completely verify that the operators work as wanted, it gives an indication that we have not lost any grid points and the total mass of the charge density should be conserved.

A.1.2 Finite difference

The finite difference operators is tested by setting up a test field based on a polynomial on which the operator should give an exact answer for. For example if we have a quantity $f(x) = 3x$, then a first order finite difference scheme will give $\hat{\nabla} f(x) = 3$.

A.1.3 Multigrid and Grid structure

We want the basic grid to be available through a grid datastructure and the stack of grids stored in the multigrid structure. To ensure that this will still work through changes in the the structs there is a simple unittest that uses a grid struct to set up a field, then it is changed in the multigrid struct. Then it confirms that the values in the grid struct is also changed.

A.1.4 Edge Operations

In the communication between the subdomains, as well as in the treatment of boundary conditions, there is a group of functions dealing with slice operations. These are tested by putting assigning each subdomain different constant values, then different slice operations is performed.

Appendix A

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) (Sbalzarini et al., 2006), Hypre (Falgout and Yang, 2002), Muelu (??), METIS (A fast and high quality multilevel scheme for partitioning irregular graphs — Karypis Lab 2016) and PETCs (manual.pdf 2016) 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

A.1 Libraries

A.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 et al., 2000). 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

A.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).

A.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.

A.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 (A fast and high quality multilevel scheme for partitioning irregular graphs — Karypis Lab 2016). 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.

A.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.

Appendix B

Code

B.0.1 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 B.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 B.2: Implementation of an recursive V-cycle

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 B.3: Codesnippet for the Z Y and X sweeps

```
  //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 B.4: 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 B.5: The foor loop doing the calculations

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

B.1 Iterative solvers

B.1.1 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 B.6: Code snippet 2D jacobian

B.1.2 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 B.7: GS-RB with if-tests

B.1.3 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 B.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;{subfigure}
      }
    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 B.9: loop routine

Bibliography

A fast and high quality multilevel scheme for partitioning irregular graphs — Karypis Lab (2016). URL: http://glaros.dtc.umn.edu/gkhome/node/107 (visited on 07/08/2016).

Birdsall, C. K. and A. B. Langdon (2004). Plasma Physics via Computer Simulation. en. CRC Press. ISBN: 978-1-4822-6306-0.

Chen, F. F. (1984). Introduction to Plasma Physics and Controlled Fusion. en. Boston, MA: Springer US. ISBN: 978-1-4419-3201-3 978-1-4757-5595-4. URL: http://link.springer.com/10.1007/978-1-4757-5595-4 (visited on 07/12/2016).

Falgout, R. D. and U. M. Yang (2002). "hypre: A Library of High Performance Preconditioners". en. In: Computational Science — ICCS 2002. Ed. by P. M. A. Sloot et al. Lecture Notes in Computer Science 2331. DOI: 10.1007/3-540-47789-6_66. Springer Berlin Heidelberg, pp. 632–641. ISBN: 978-3-540-43594-5 978-3-540-47789-1. URL: http://link.springer.com/chapter/10.1007/3-540-47789-6_66 (visited on 07/08/2016).

Fitzpatrick, R. (2014). Plasma Physics: An Introduction. English. 1 edition. Boca Raton: CRC Press. ISBN: 978-1-4665-9426-5.

Goldston, R. J. and P. H. Rutherford (1995). Introduction to Plasma Physics. en. CRC Press. ISBN: 978-1-4398-2207-4.

manual.pdf (2016). URL: http://www.mcs.anl.gov/petsc/petsc-current/docs/manual.pdf (visited on 07/08/2016).

Pécseli, H. L. (2012). Waves and Oscillations in Plasmas. English. 1 edition. Boca Raton: CRC Press. ISBN: 978-1-4398-7848-4.

Press, W. H. et al. (1988). "Numerical recipes in C". In: Cambridge University Press 1, p. 3. URL: http://journals.cambridge.org/abstract_S0269964800000565 (visited on 04/21/2016).

Sbalzarini, I. et al. (2006). "PPM – A highly efficient parallel particle–mesh library for the simulation of continuum systems". en. In: Journal of Computational Physics 215.2, pp. 566–588. ISSN: 00219991. DOI: 10.1016/j.jcp.2005.11.017. URL: http://linkinghub.elsevier.com/retrieve/pii/S002199910500505X (visited on 07/08/2016).

Trottenberg, U. et al. (2000). Multigrid. Academic press. URL: https://www.google.com/books?hl=en&lr=&id=9ysyNPZoR24C&oi=fnd&pg=PP1&dq=

trottenberg+2000&ots=rJCHSPzSMY&sig=sin3i-gmWOoykoTFnyHGIPZXT5Q (visited on 04/21/2016).