FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT • DEPARTMENT INFORMATIK

Lehrstuhl für Informatik 10 (Systemsimulation)

Geometric multigrid for the gyrokinetic Poisson equation from fusion plasma applications

Christina Schwarz

master thesis

Geometric multigrid for the gyrokinetic Poisson equation from fusion plasma applications

Christina Schwarz

master thesis

Aufgabensteller:	Prof. Dr. U. Rüde
Betreuer:	Philippe Leleux, Martin Kühn
Bearbeitungszeitraum:	15.06.2021 – 15.12.2021

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Systemsimulation (Informatik 10), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der master thesis einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 14. Dezember 2021 ....

Contents

1 Introduction **1**

2 Plasma fusion for energy creation **2**
2.1 What is plasma fusion? 2
2.2 Magnetic confinement of particles in tokamaks 3
2.3 Current research and projects 6

3 Plasma simulation with GyselaX **8**
3.1 Gyrokinetic code 8
3.2 Project EoCoE 9

4 Geometric polar multigrid for a 2D Poisson-like equation with implicit extrapolation **11**
4.1 The given problem: a 2D quasi-neutrality Poisson-like equation 11
4.2 The finite difference discretization of the Poisson equation on disk-like domains . . . 13
4.3 Introduction to Multigrid 16
4.3.1 Direct vs. iterative solvers 16
4.3.2 The 2-level multigrid 18
4.3.3 The multigrid V-cycle 19
4.4 Two problems arising with the choice of polar coordinates 21
4.4.1 Singularity at the origin 21
4.4.2 Anisotropy 21
4.5 Multigrid with implicit extrapolation 27

5 Implementation of the solver GmgPolar **34**
5.1 Structure of the code 34
5.1.1 Namespace param 35
5.1.2 Namespace gyro 35
5.1.3 Class gmgpolar 36
5.1.4 Class level 37
5.2 Implementation of the multigrid cycle 38
5.2.1 Matrix-free implementation 39
5.2.2 Problem setup 40
5.2.3 The multigrid cycle itself 44
5.3 Numerical experiments 49

6 Improving the solver GmgPolar **51**
6.1 Code optimization 51
6.2 Decoupled circle-radial smoothing 53
6.3 Implicit extrapolation with full grid smoothing 55
6.4 Parallelisation 56
6.4.1 Parallelisation with OpenMP 57
6.4.2 Parallelisation with GPUs and Cuda 60
6.5 Further optimizations 63

7 Conclusion **64**

List of Figures

1 Binding energy per nucleon for the atoms in the periodic table; by *Wikipedia Commons. (Left)* Elements for fusion reactions, e.g. the fusion of deuterium-tritium into helium is represented, generating energy ΔE. *(Right)* Elements for fission reactions, e.g. U^{235} is split into Kr^{89}. 2
2 Toroidal magnetic confinement of charged particles; republished with permission of *John Wiley & Sons - Books*, from *Fusion Plasma Physics, Stacey, W. M., 2005*; permission conveyed through Copyright Clearance Center, Inc. [61]. 4
3 Fusion reactors with toroidal magnetic confinement; reprinted with permission of *The Economist* [20]. 4
4 Creation of the magnetic fields with external coils; reproduced with permission of the Licensor through PLSclear, from *Tokamaks, Wesson, J., 1997*, John Wesson 1987, 1997 [66]. 5
5 Induction of a current in the plasma; reproduced with permission of the Licensor through PLSclear, from *Tokamaks, Wesson, J., 1997*, copyright John Wesson 1987, 1997 [66]. 6
6 Concept of a tokamak with spirlaing trajectory of the particles around the helical magnetic field; reprinted with permission of *Eurofusion* [24]. 6
7 International project ITER in France. 7
8 Schematic development of the 2D Poisson equation from a 6D Vlasov equation. . . . 8
9 Schematic view of the GyselaX code where a 2D Poisson equation must be solved at each time step [26]; courtesy of *V. Grandgirard*. 9
10 Variation of the diffusivity coefficient α over the radius r. 11
11 Mapping of the grid from Cartesian (Ω) to polar coordinates ($\tilde{\Omega}$), for the inverse mapping we have to exclude the origin ($r_0 > 0$) to avoid a singularity 12
12 Different geometries for the poloidal cross-section. 13
13 Schema of a multigrid V-cycle with N grids, denoting the finest level by 0. The final approximation of the solution after the V-cycle u_0^* is more accurate than u_0 from the start. 19
14 An example grid of size 4×4, using periodicity conditions in θ-direction. The indicated nodes are smoothed simultaneously, i.e. line relaxation in resp. circle and radial direction. 22
15 Different zebra line smoothers. 24
16 An example grid of size 4×7, using a combination of circle and radial smoother. . . 25
17 System matrix corresponding to the example grid of size 4×7: The red line indicates the separation between circle and radial smoother. Sections of the matrix corresponding to the four matrices A_{sc} are coloured in either black or white. The remaining elements correspond to the matrices $A_{sc}^{\perp}$. 26
18 The four different bases: linear, h-hierarchical, p-hierarchical and quadratical, functions on the fine grid are marked in violet and functions on the coarse grid in red. . 29
19 The already presented example grid of size 4×7 (compare to Figure 16) with the elimination of the coarse nodes for the smoothing procedure in case of implicit extrapolation. As a consequence, the node numbering changes in comparison to the previous example. 32
20 Extrapolated smoothing matrix corresponding to the example grid of size 4×7 in Figure 19. The elimination of the matrix rows corresponding to the coarse nodes is visualized with green lines. The red arrows indicate the matrix columns corresponding to the coarse DOFs, with the red circles showing all elements which have to be shifted from the matrix A_{sc}^{ex} to $A_{sc}^{ex,\perp}$. 33
21 Structure of the GmgPolar C++ implementation. The class *gmgpolar* contains several instance of the class *level*, then the colors represent calls of methods in *level* from the methods in *gmgpolar*. 34
22 Anisotropic refinement of the grid around $r_\alpha = 2/3R$ in the r-direction. 40
23 Stencil representation for 4 different types of nodes, depending on their position with respect to the neighbouring coarse nodes [36]. 41
24 Example 4×7 grid with both global numbering and numbering local to each smoother. 42

25 Stencils of the matrices A_{sc} and $A_{sc}^{\perp}$ for each smoother in the polar plane [22]. . . . 43
26 Principle behind the application of the operator A. In the original version, for each point we compute the coefficient of neighbouring points to update the local operator entries. In the optimised version, for each point, the local coefficients are computed to update the operator entries of neighbouring points. 52
27 Comparison of the run time of the method *level::apply_A* of the original and the optimised version. 52
28 Conflicts in red cross resulting from the parallel update of two consecutive radii r_i and r_{i+1} in the function *apply_A*. 57
29 Splitting of the workload in three different parts, where each task group is independent. 58
30 Dependencies of the OpenMP tasks. The relations are read from bottom to top. . . 58
31 Speed-up of the function *level::apply_A* using OpenMP parallelisation for two different grid sizes m. 60

List of Tables

1 Parameters of the namespace *param*. 35
2 Functions from namespace *gyro*. 36
3 Variables from the class *gmgpolar*. 36
4 Functions from class *gmgpolar*. 36
5 Variables from class *level*. 37
6 Functions from class *level*. 38
7 Identification of the smoother for a node i/j. 42
8 Mapping from global coordinates (i, j) in the grid, to local coordinates (row, col) for all smoothers. 42
9 Application of the stencil to build the matrix A_{sc}. *PB*: indicates that a node lies on the periodic boundary. *AOD*: denotes the across-the-origin discretization. $nt = ntheta_int$. 43
10 Application of the stencil to apply the matrix $A_{sc}^{\perp}$. PB: indicates that a node lies on the periodic boundary. BRI: *base_row_index*. $nt = ntheta_int$, and $dc = delete_circles$. 46
11 Mapping from the local index k of a vector of the smoother sc, to global coordinates (i, j) in the grid. $nt = ntheta_int$, $dc = delete_circles$. 47
12 Mapping from global coordinates (i, j) in the grid, to local coordinates (row, col) for the smoother *sc* when the implicit extrapolation is active on the finest level. 47
13 Results of the multigrid solver, for a test problem with Dirichlet boundary conditions on the inner radius (10^{-5}) and outer radius for the Shafranov geometry. Iteration count *its*, mean residual reduction factor $\widehat{\rho}$, errors of the iterative to exact solution evaluated at the nodes in the weighted 2- and ∞-norms with corresponding error reduction order. 50
14 Run time (in s) of the code versions for test problems of increasing size using the Shafranov geometry, with Dirichlet boundary conditions on the inner radius ($r_0 = 10^{-5}$) and outer radius. 53
15 Iteration count and run times when applying GmgPolar with the two different smoothing procedures, for a test problem of increasing sizes using the Shafranov geometry, with Dirichlet boundary conditions on the inner radius ($r_0 = 10^{-5}$) and outer radius. 54
16 Iteration count and error norms of the different extrapolation variants, for a test problem with Dirichlet boundary conditions on the inner radius (10^{-5}) and outer radius for the Shafranov geometry, using the circle-radial coupled smoothing version. Notations as in Table 13. 56
17 Comparison of the runtime in seconds of the sequential CPU and the parallel GPU version. 62

1 Introduction

In order to face climate change and to preserve our ecosystem, we have to reduce the overall emission of carbon dioxide into the atmosphere by omitting the use of fossil fuels. In the urgent context of energy transition, we mostly talk about renewable energies, but there exists another promising alternative for a CO_2-neutral energy source.

Nuclear fusion, also taking place in the sun's interior, is the reversed nuclear fission process and the only known energy source on earth not yet harnessed [42]. In contrast to the fission process, nuclear fusion is safe and emits only short-lived nuclear waste. Delivering an almost infinite amount of clean energy and with almost inexhaustible resources on earth, the plasma fusion would solve all the world's climate and energy problems [15]. However, being extremely complex and extraordinarily challenging from a scientific point of view [69], the reaction cannot be maintained for sufficient long time, yet, for a satisfactory energy balance, as it is extremely unstable. For many years, multiple scientists and researchers have been already working on the replication of the sun's fusion process on earth with the goal of finally converting the produced heat into electrical energy [69].

As the construction and operation of fusion reactors, e.g. tokamaks, is exceptionally expensive, and thus real-life experiments are in many cases impossible, numerical simulations are required in order to gain more knowledge about the fusion process and to improve the reactor conception. One existing code for plasma simulations in a tokamak is called GyselaX, in which a five dimensional Vlasov-equation and a three dimensional Poisson-equation need to be solved. The latter one may be reduced to the repeated solution of a two dimensional Poisson-like equation on many disk-like cross-sections of the tokamak geometry. In this context, the EoCoE (Energy Oriented Center of Excellence) project, funded by the European Commission, aims for the improvement of the current solver for the 2D equation in order to reduce the simulation time of GyselaX.

In [38, 39], a geometric multigrid approach has been developed, using finite differences for the discretization. Due to a given variable coefficient of the problem and the usage of generalised polar coordinates, some challenging difficulties arise. To overcome the introduced singularity at the origin and to handle the anisotropy of the grid, different approaches are possible. Additionally, an implicit extrapolation technique is proposed to increase the approximation order of the solution. In this master's thesis, the multigrid solver, called *GmgPolar*, shall be implemented in a matrix-free manner in C++.

The thesis is structured as follows. In Sections 2 and 3, the physical basics of plasma fusion and the mathematical description for the simulations are explained. After outlining the goal of the underlying project EoCoE, the given problem and our approach to solve it with geometric polar multigrid and implicit extrapolation are described, as proposed in [38, 39], in Section 4. Finally, the implementation of the solver *GmgPolar* as well as several improvements and optimizations to the code are presented in the Sections 5 and 6.

2 Plasma fusion for energy creation

2.1 What is plasma fusion?

It is well known that the sun converts its mass into energy according to Einstein's famous law $E = mc^2$. This process is called *plasma fusion* or *nuclear fusion*. Based on [61, chapt. 1] and [66, chapt. 1], in this section, we introduce the principles of nuclear reactions, where the binding energy of atoms is released.

The actual mass of an atomic nucleus is not equal to the sum of the masses of its separate nucleons, i.e. its protons and neutrons, of which it is composed. The difference, the so called *mass defect* Δm, is converted into energy when the nucleus is formed. The amount of externally supplied energy necessary to disassemble a nucleus into its separate nucleons, we call the *binding energy* per nucleus. Then, by converting atoms with low binding energy into atoms with higher binding energy, kinetic energy is released. The binding energy for all elements of the periodic table is given in Figure 1. On the left side of the diagram, the binding energy increases with the number of nucleons. Hence, nuclear energy can be released through the combination of these small atoms, i.e. with the fusion process. Whereas, on the right side of the diagram, the binding energy decreases with the number of nucleons and thus, the fission process can be used to release energy by splitting large atoms.

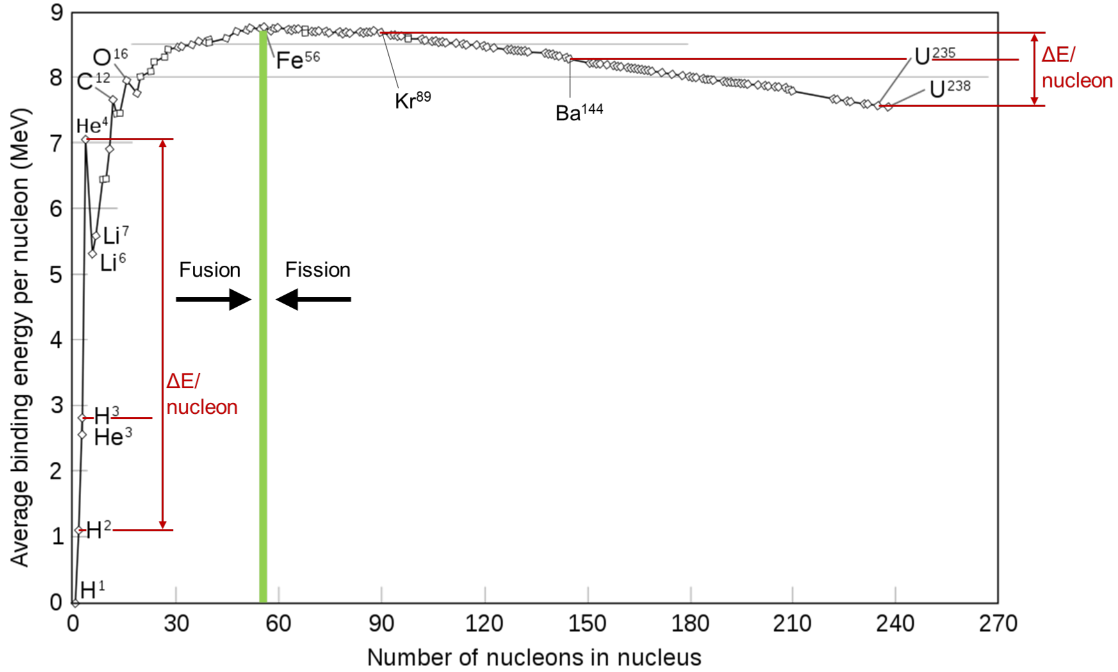


Figure 1: Binding energy per nucleon for the atoms in the periodic table; by *Wikipedia Commons*. *(Left)* Elements for fusion reactions, e.g. the fusion of deuterium-tritium into helium is represented, generating energy ΔE. *(Right)* Elements for fission reactions, e.g. U^{235} is split into Kr^{89}.

Let's compare the energy released from fusion and fission with a simple example taken from [66, chapt. 1]. The most promising fusion reaction is that of the nuclei deuterium and tritium (hydrogen isotopes, H_1^2 and H_1^3) which fuse to an alpha particle, i.e. an helium nucleus, with the release of a neutron n_0^1. We then have the reaction

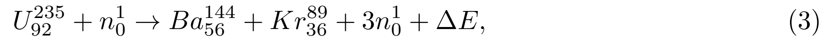
$$H_1^2 + H_1^3 \rightarrow He_2^4 + n_0^1 + \Delta E, \qquad (1)$$

where ΔE is the energy, corresponding to a mass deficit of $0.01888286\, u$, released as kinetic energy of the reaction products

$$\Delta E = \Delta m \cdot c^2 = 0.0188862\, u \cdot c^2 = 2.819 \cdot 10^{-11}\, J = 17.592\, MeV, \qquad (2)$$

with $1u = 1.660539 \cdot 10^{-27}\,kg$ being the unified atomic mass unit.

The most common fission reaction is that of Uranium-235 [32, 25], defined as



$$U_{92}^{235} + n_0^1 \rightarrow Ba_{56}^{144} + Kr_{36}^{89} + 3n_0^1 + \Delta E, \tag{3}$$

where the energy released in the fission process is

$$\Delta E = \Delta m \cdot c^2 = 0.186017\,u \cdot c^2 \approx 2.776 \cdot 10^{-11}\,J = 173.273\,MeV. \tag{4}$$

Taking into account that the mass of the reactants is 50 times larger in this fission reaction compared to the fusion reaction, the energy released from fission is significantly inferior. Moreover, deuterium, which is used for the nuclear fusion, is a plentiful resource on earth and contained in sea water. The radioactive tritium though, does not occur naturally, but can be bred from lithium, of which there are large reserves, using the released neutrons. In fact, just $1\,kg$ of this fuel would release about $10^8\,kWh$ of energy and would provide the requirements of $1\,GW$ electrical power station for a day. Additionally to its relatively lower energy release, the fission process is quite dangerous since it is explosive and thus implies the risk of a core meltdown. In contrast, the fusion process is quite safe as the instability of the created plasma breaks the reaction at the smallest disturbance with no risk of a chain reaction or meltdown. Also less dangerous radioactive waste is produced, as the alpha-radiation (positively charged helium particles) is less long-lived than the gamma-radiation from fission. Moreover, the most common reactants, deuterium and tritium, are available in large quantities and almost inexhaustible on earth [69]. Consequently, the realization of nuclear fusion would be an important step within the energy transition towards carbon neutrality as it is an inexhaustible source of clean, green energy. However, in practice, the fusion process is quite complex to realize on earth and not yet possible to be used for energy creation [1, 15].

In order for the fusion reaction to take place, the two involved nuclei must overcome the long-range electrostatic, or Coulomb, repulsion force due to their positive charges, and approach sufficiently close so that the short-range nuclear attraction forces become predominant, and the formation of a compound nucleus becomes possible. The chance for a fusion of two nuclei increases with the atoms velocity, and thus, with the temperature [15]. Owing to this, the authors of [61, chapt. 1] state that high energies are required of the order of $10\,keV$ to $100\,keV$, corresponding to thermonuclear temperatures of $108\,K$ to $109\,K$ which are comparable to those of the sun's interior. In these conditions, light atoms are entirely stripped of their orbital electrons and we obtain an ionized but macroscopically neutral gas, composed of positively charged atomic nuclei and electrons. This type of gas is called a *plasma*. Whereas the sun has about 333 000 times the mass of our planet and consequently an extremely high pressure ($150\,g/cm^3$) in its core, it is not possible to meet such consitions on earth [42]. Consequently, to achieve the same amount of energy as in the sun's core, temperatures about six times the sun temperature (100 to $150 \cdot 10^6$ K) are necessary for fusion reactions to take place.

Controlled fusion reactions to produce energy are in principle obtained as follows. First, the plasma is heated by radio frequency heating, neutral beam injection and/or ohmic heating. When heated to thermonuclear conditions, the released alpha particles, i.e. helium nuclei, then transfer their energy to the plasma through collisions [1], thus providing an increasingly large fraction of the total heating. The event, when the plasma temperature can be maintained solely by internal alpha-particle heating and the burning process becomes self-sustained without any further applied external heating, is called ignition. 80 % of the reaction energy is carried by the neutrons, which are channeled via a surrounding blanket and transformed into heat, i.e. the reactor power output. See [66, chapt. 1] for practical details on this processes in fusion reactors.

In the end, the challenge of plasma fusion is to heat the plasma to sufficiently high temperature and confine it for a sufficiently long time. In order to get a positive energy balance, the thermonuclear energy produced needs to significantly exceed the energy required to heat the plasma, and thus, the charged particles must retain their energy and remain in the plasma for sufficient time.

2.2 Magnetic confinement of particles in tokamaks

The plasma, being heated to thermonuclear temperatures, would immediately vaporize the reactor walls upon direct contact. Therefore, another method of confinement is needed for the plasma, namely magnetic confinement, which is explained in detail in [66, chapt. 2,4] and [61, chapt. 2,3].

Charged particles, including nuclei and electrons in plasma, spiral around magnetic field lines due to the Lorentz force which deflects them perpendicular to the field and their own velocity. Since the plasma particles are free to move uniformly along the field lines but constrained in their perpendicular motion, they spiral around the field lines with a radius which is inversely proportional to the strength of the magnetic field. Theoretically, a magnetic field may then be configured in a circular shape in order to confine the particles in a toroidal region, gyrating endlessly around the magnetic orbits [1], see Figure 2.

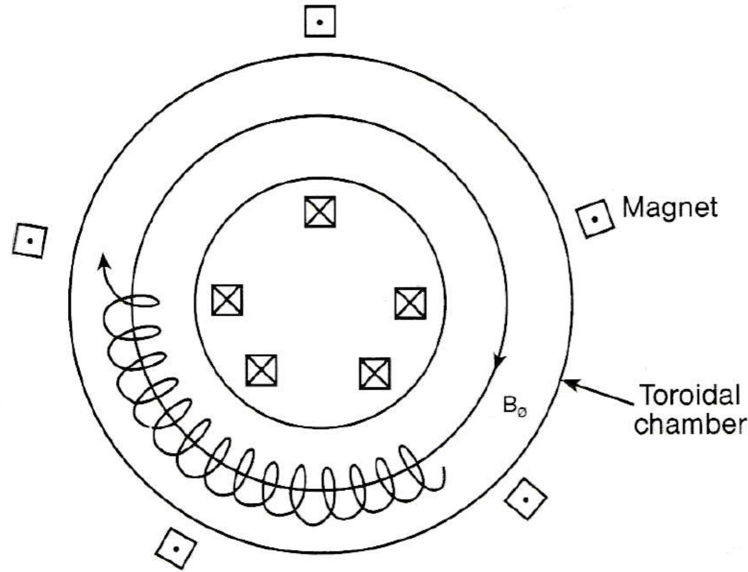


Figure 2: Toroidal magnetic confinement of charged particles; republished with permission of *John Wiley & Sons - Books*, from *Fusion Plasma Physics, Stacey, W. M., 2005*; permission conveyed through Copyright Clearance Center, Inc. [61].

There are two main types of such toroidal plasma confinement systems using vacuum chambers for the plasma: the tokamak and the stellarator, which differ mainly in the shape of the magnetic coils used to create the strong external magnetic fields [69].

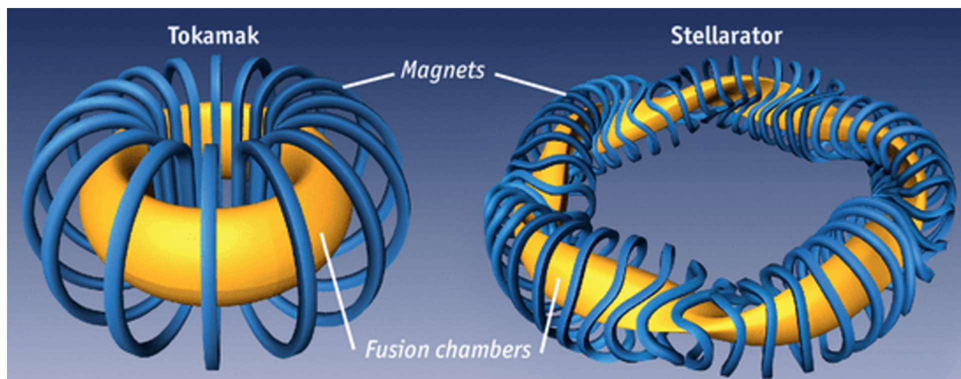


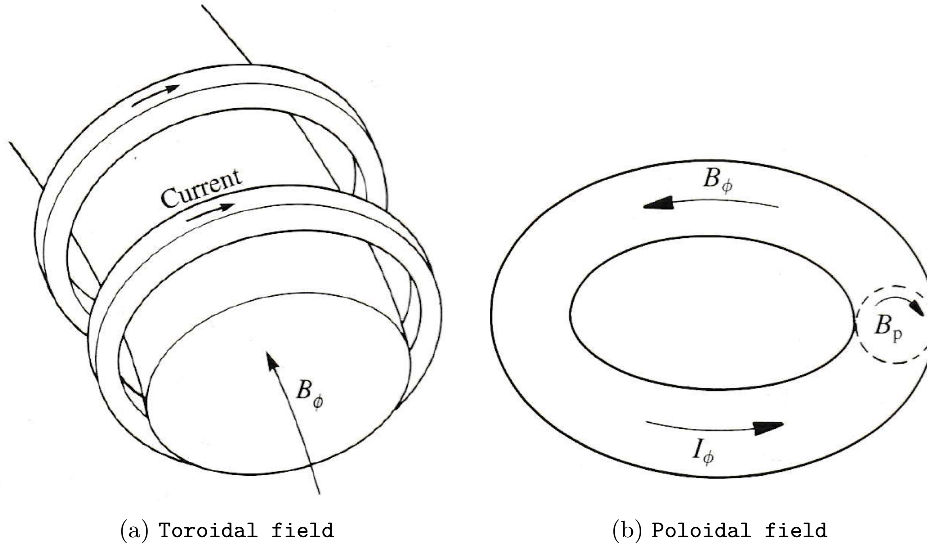
Figure 3: Fusion reactors with toroidal magnetic confinement; reprinted with permission of *The Economist* [20].

In a stellarator, see Figure 3 (right), the whole magnetic field is constructed by external coils. On the contrary, tokamak reactors, see Figure 3 (left), use a current in the plasma additionally to external coils. As a result, tokamaks are much easier to understand and to control, because

they only have 2D coils instead of 3D coils [15]. The device has been invented in the USSR in the mid-1960s and the word *'tokamak'* is derived from the Russian words *'toroidalnaya kamera'* and *'magnitnaya katushka'*, meaning *'toroidal chamber'* and *'magnetic coil'*. Tokamaks are the most advanced and widely investigated type of fusion reactors worldwide.

Tokamaks use a combination of different magnetic fields to create the confinement of particles in a toroidal region, with a D-shaped cross-section. The main magnetic field is the toroidal field. It is produced by poloidal currents in external *toroidal field coils* which encircle the plasma, see Figure 4a. However, in order to keep all particles within the toroidal shape, the magnetic field has to be stronger on the inside of the torus than on the outside due the smaller inner radius. This non-uniformity of the magnetic field produces forces which act upon the charged particles to drift outward - perpendicular to the magnetic field - to low field regions. In fact, curved magnetic fields tend to straighten themselves and any deviation from a uniform field, such as field gradients, curvature or polarization, leads to particle drifts. Those drifts then carry positive and negative particles differently, causing charge separation which leads to additional electric fields and instabilities of the principal magnetic field [42, 1].

In order to compensate these drifts and to have an equilibrium in which the plasma pressure is balanced by all magnetic forces, a poloidal magnetic field must be superimposed upon the toroidal field. This poloidal field is typically produced by a toroidal current flowing in the plasma, see Figure 4b. The current is induced by a transformer action of a set of primary coils in the centre of the torus, see Figure 5a. Hereby, a change in the magnetic flux through the torus, corresponding to the primary winding, induces a toroidal electric field which then drives the toroidal current, see Figure 5b. Ideally, this current should be continuous in time. However, since the driving electrical field is induced by an increasing magnetic flux, this can only be continued for a limited period, possibly one hour. Consequently, a tokamak reactor can only be operated in a pulsed behaviour. In contrast to this, a stellarator produces the poloidal field by external coils and hence is able to operate continuously.

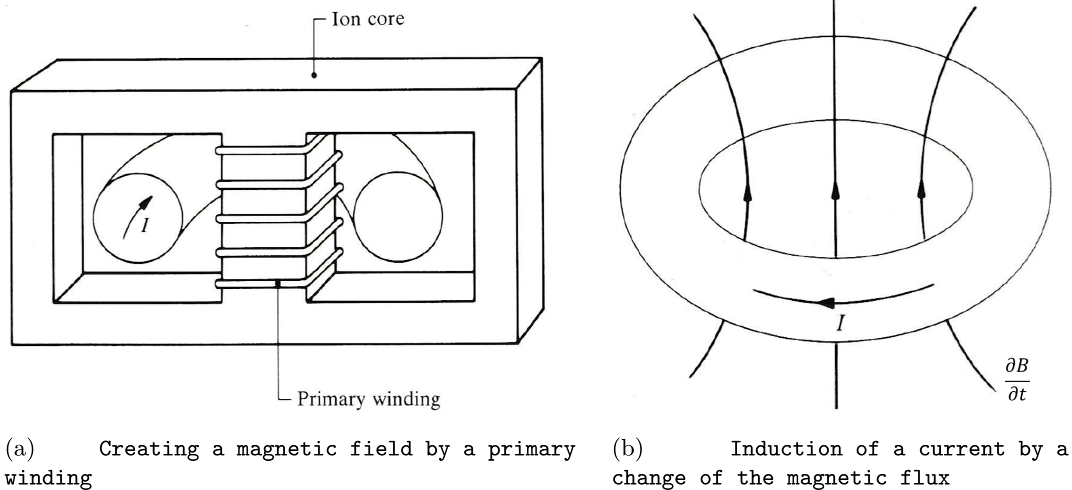


(a) `Toroidal field` (b) `Poloidal field`

Figure 4: Creation of the magnetic fields with external coils; reproduced with permission of the Licensor through PLSclear, from *Tokamaks, Wesson, J., 1997*, John Wesson 1987, 1997 [66].

At last, combining the toroidal and the poloidal field results in a magnetic field of the form of a helical trajectory around the torus. As a consequence, the charged plasma particles gyrate around the helical magnetic field lines, see Figure 6. Additionally to this complex motion of the particles, there occur multiple, complicated, yet unexplored disturbances within the plasma, which make all predictions and computations even more difficult, e.g. particle interactions, inhomogenities, instabilities or turbulence.

<u>Remark:</u> To point out the difference of a tokamak to a particle accelerator, in a tokamak, the particles are not accelerated all into the same direction. On the contrary, from the prevailing high



(a) Creating a magnetic field by a primary winding

(b) Induction of a current by a change of the magnetic flux

Figure 5: Induction of a current in the plasma; reproduced with permission of the Licensor through PLSclear, from *Tokamaks, Wesson, J., 1997*, copyright John Wesson 1987, 1997 [66].

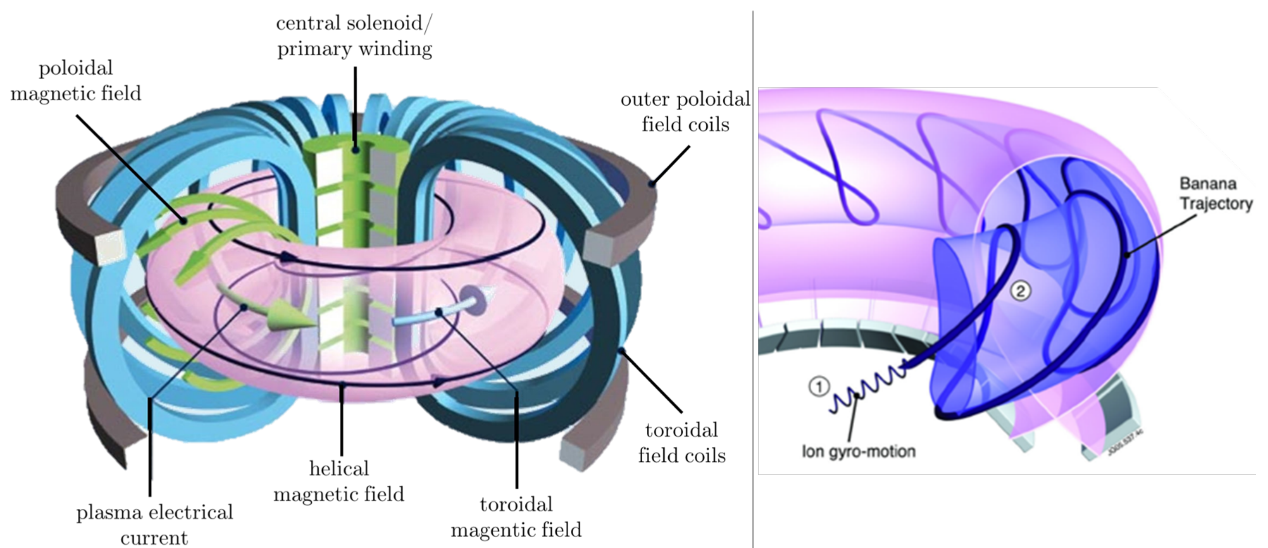


Figure 6: Concept of a tokamak with spirlaing trajectory of the particles around the helical magnetic field; reprinted with permission of *Eurofusion* [24].

temperature follows that the particles' velocities and moving directions are defined by a Gaussian distribution. Besides, the differences in mass and charge of the particles lead to different refraction and drifts within the torus [60].

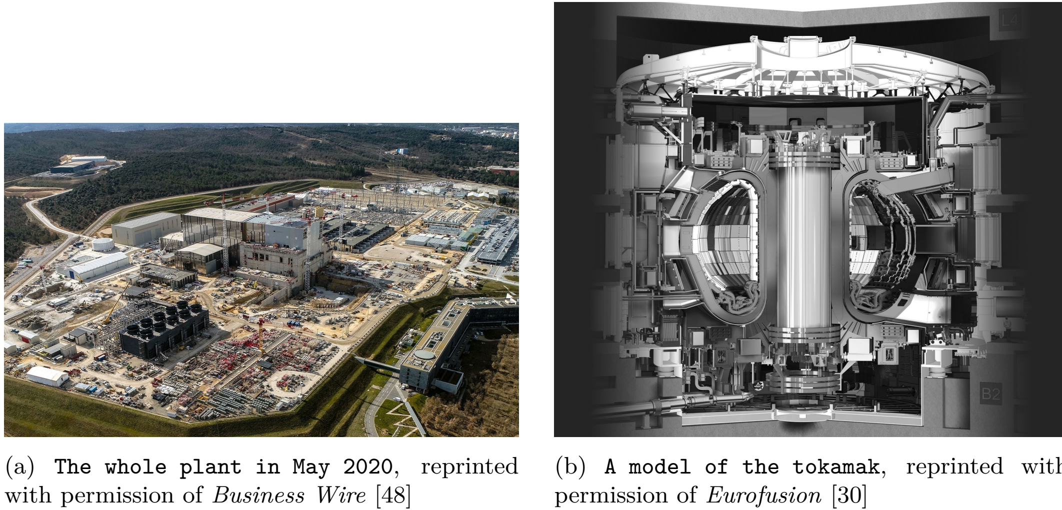
2.3 Current research and projects

So far, the reactions in a tokamak are not completely understood yet, as they are extremely complex. Due to the limits of the magnetic field, such as impurities of the plasma, thermal losses being caused by conduction, convection or radiation, and massive instabilities, the largest problem is to permanently maintain the rather unstable plasma current. In fact, the maximum confinement time attained is still way too short and thus no satisfactory energy balance could be achieved, yet [66].

One of the most promising projects is the HL-2M tokamak in Chengdu, China, which was finished in December 2020. The so called *artificial sun* achieves temperatures up to 150 million degrees and was able to attain the current world record with a fusion process maintained during 10 seconds [15].

Another promising project is ITER (International Thermonuclear Experimental Reactor), a

collaboration of 35 states worldwide [1], see Figure 7. The ITER agreement was signed in the year 2006, the construction in Cadarache in Southern France started in 2010 and the first plasma is currently planned for 2025 [30, 42].



(a) **`The whole plant in May 2020`**, reprinted with permission of *Business Wire* [48]

(b) **`A model of the tokamak`**, reprinted with permission of *Eurofusion* [30]

Figure 7: International project ITER in France.

The construction of this world's largest tokamak is extremely complex [1]:

- The 18 toroidal field coils are among the largest and most precise magnets that were ever build.
- To improve efficiency they are superconducting in order to have less electrical resistance and to carry higher current while consuming less power.
- The necessary Helium-cooling down to almost the absolute zero-point ($4\,K$) is in fact an immense challenge, while temperatures of about 150 million degrees prevail in the inner of the tokamak.
- Extra research projects are carried out to investigate for example the welding work or to improve autonomous robots for the exchange of modules in the radioactive inner of the reactor.
- Due to the pulsed behaviour of the tokamak, about one hour is necessary between the pulses for the system to restart [42].

As a consequence, the project is the world's most expensive science project of all time. The initial budget of the experimental reactor was estimated at 6 billion dollars, but the total cost is now estimated at around 22 billion dollars or even higher [17].

The principle question is whether a tokamak reactor will ever be able to create a self-sustained fusion reaction to actually produce energy and thus be competitive with other power systems, such as fossil fuels, someday. Even though power outputs of up to 10% of the input power have been already reached in 1997 with the *JET* tokamak [66], it is still a long way to go until the commercial use of fusion power. Since 2018, the current world record is hold by the *Japanese Torus (JT-60)* with a total positive energy output of 20 % [30, 47], however still not enough for ignition. It has to be indicated that all existing reactors, including ITER, are only experimental devices and meant to conduct long-term research on how to overcome all previous listed issues in order to improve the fusion process. Similarly, in the field of the stellarator, so for example at the Max-Planck-Institute for plasma physics in Greifswald, Germany, at the *Wendelstein 7X* reactor, only basic research is conducted, yet [42].

Despite all the involved issues, China announced to be aiming for the first industrial prototype in 2035 right after the first successful operation of HL-2M tokamak [15]. The professor of physics at the Tsinghua University in Peking Gao Zhe stated:

> "There is no guarantee that all these problems will be solved. But if we don't do it, the problems will definitely not be solved" [15].

3 Plasma simulation with GyselaX

As fusion reactors are very expensive and complex, real-life experiments are mostly impossible and numerical simulations are required to increase our knowledge about fusion plasma and how to improve tokamaks. Therefore, a mathematical description of the plasma particles is necessary. Among all the possible models for the simulation of plasma, the so-called *gyrokinetic* theory is the theoretical framework chosen in the *GyselaX* code to study plasma behavior [26].

According to Bouzat et al. in [8], gygrokinetic simulations are able to capture the turbulent transport of ions and electrons which is a limitation for the performance of fusion reactors. However, efficient and robust numerical schemes, accurate geometric descriptions, as well as efficient HPC (*high performance computing*) techniques and good parallelisation algorithms are required to reduce the immense simulation time and thus the expensive cost of such simulations.

3.1 Gyrokinetic code

The six dimensional gyrokinetic Vlasov equation describes the time evolution of the distribution function of plasma particles [26], meaning "the probability of finding a particle at a given position, with a given velocity [...] at a given time" [69]. The differential equation defines the toroidal geometry as a three dimensional space along with the particle's velocity in three supplementary dimensions [8] and is solved for each ion species [26].

As we are mainly interested in the turbulent fluctuations, we can neglect high-frequency dynamics, corresponding mostly to the gyromotion (i.e. the fast spiraling motion around the magnetic field lines), and focus on the lower frequencies expressing the slower drift motions of interest [26]. The higher frequencies are removed via the so-called gyroaverage operator and a phase-space coordinate transformation, see [26, 69] for details, and the guiding center distribution is transformed into the actual particle distribution, reducing the 6D problem to a 5D one [8].

Furthermore, the authors in [40] state that in tokamak configurations, we can assume the overall quasi-neutrality of plasma, even though particles are locally charged. Hence, different ion species are ignored and inertia of electrons, magnetic fluctuations and particle collisions are neglected [71]. To express this idea in the gyrokinetic code *GyselaX* [8], the 5D Vlasov equation is non-linearly and self-consistently [26] coupled to a 3D quasi-neutrality Poisson-like equation. This 3D solver computes the electric field or potential that corresponds to the particle distribution at each time step, and can be written in dimensionless variables [40]. The operator of this equation is tightly coupled to the geometry in the poloidal plane, i.e. perpendicular to the the magnetic field lines, and can be decoupled from the periodic toroidal direction, which is aligned with the field lines, via a Fourier projection [8]. Thus, the three dimensional quasi-neutrality equation can be reduced to a 2D equation expressed on the poloidal cross-section of the torus, with inseparable radial (r) and poloidal (θ) dimensions. At each time step of the simulation, an instance of the resulting 2D Poisson equation must be solved, i.e. with the help of finite elements or differences in *GyselaX* [8, 26]. Figure 8 summarizes this reduction of dimensions for the Vlasov and quasi-neutrality equations.

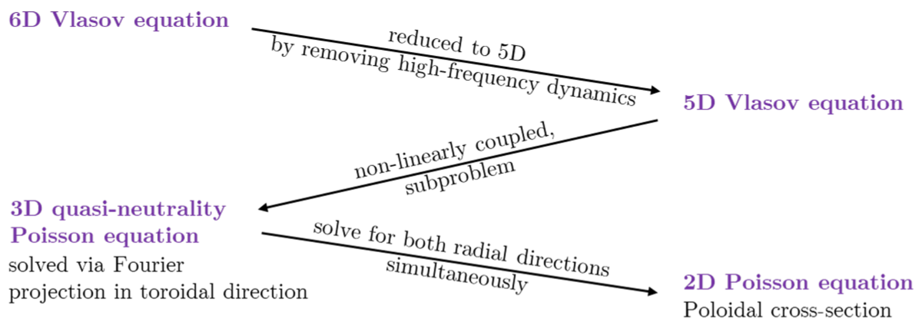


Figure 8: Schematic development of the 2D Poisson equation from a 6D Vlasov equation.

As stated in [26], the electrostatic code *GyselaX* (gyrokinetic semi-Lagrangian) "is one of the few international 5D gyrokinetic codes able to perform global, full-f and flux-driven simulations".

In other words, not only perturbations with respect to some prescribed background equilibrium are computed, but the whole distribution function is evolved and long time behavior of plasma turbulence and transport is explored. The code is based on a semi-Lagrangian method, which is a mixture between PCI (*particle-in-cell*), see [41], and Eulerian approaches, see [12], and where the grid points of a discretization move along the characteristic trajectories of the transport equation [69].

GyselaX solves the described 5D Vlasov equation coupled with the reduced 3D quasi-neutrality equation [23] as described before. The code is parallelised using an hybrid OpenMP-MPI paradigm [8]. Figure 9 shows the overall flow of the GyselaX code, with the solution of the Poisson equation at each time step in a red circle, which is the focus of this master's thesis.

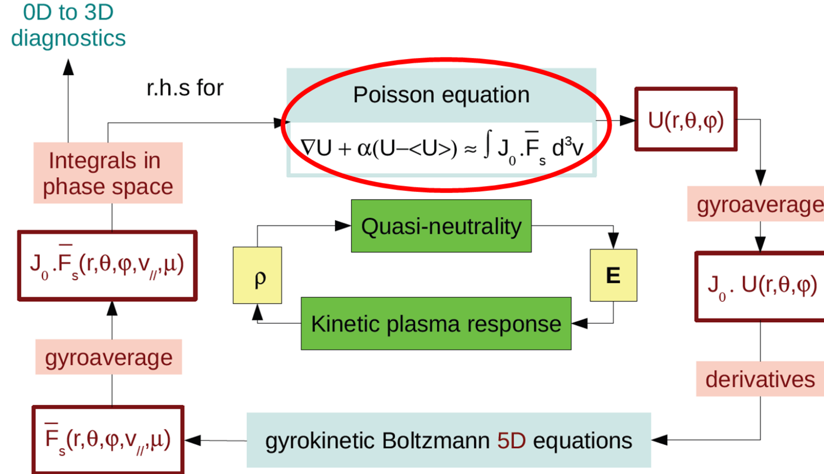


Figure 9: Schematic view of the GyselaX code where a 2D Poisson equation must be solved at each time step [26]; courtesy of *V. Grandgirard*.

3.2 Project EoCoE

The following paragraph is based on the official website [23] of the EU project EoCoE II (Energy oriented Center of Excellence: towards exascale for energy), which aims to accelerate the transition to a cheap and reliable low carbon energy supply with the help of High Performance Computing. At the crossroads of the digital revolution and energy transition, EoCoE II develops modern computational methods in order to improve the management, production and storage of clean, decarbonized energy. The main goal of the project is to create a multi-disciplinary platform, a network of experts in High Performance Computing, numerical mathematics and sustainable energies. By improving underlying models, the objective is to improve the accuracy and to reduce the number of core hours of simulations in the application areas: meteorology, materials, hydrology and fusion.

As stated in [22], solving linear algebra problems is a core task in EoCoE II scientific challenges, and thus the goal of the work package WP3 is to design and implement scalable, exascale-enabled linear algebra solvers for selected applications and to integrate them into the application codes. Regarding the simulation of magnetically confined plasma in tokamak reactors, it is intended within WP3 to improve the current in-house solver for the solution of the 2D quasi-neutrality equation in GyselaX [26], which is a bottleneck for large scale simulations being not efficient enough. For this purpose, the contributing teams - the Max-Planck-Institute in Munich, the FAU Erlangen-Nuremberg, CERFACS in Toulouse and CEA from Paris - met in November 2018 for the first time [21] and regular meetings have then taken place since the beginning of 2021 [22]. The idea is to conduct an extensive comparison between three different solvers: the framework AMRex, a spline based and a geometric polar multigrid approach. After establishing the advantages and disadvantages of each method in terms of accuracy, convergence and cost with respect to the requirements of GyselaX, one or several of these solvers can be integrated into the plasma simulation.

The first approach [35] is a geometric multigrid solver for linear systems in Cartesian coordinates, using block-structured adaptive mesh refinement, which is based on the parallel C++ software

framework AMRex [68]. The second approach [71, 70], based on a 2D guiding-center model is a combination of two solvers. The hyperbolic part of the target model equation is solved with the method of characteristics (a semi Lagrangian advection solver employing pseudo-Cartesian coordinates) and the elliptic Poisson equation with a finite element solver based on globally C^1-smooth polar B-splines [70].

The last solver is called *GmgPolar* and uses geometric multigrid in polar coordinates together with an implicit extrapolation technique [38, 39]. It was developed by the FAU in collaboration with CERFACS and is the main subject of this thesis. In the next Section 4, after a reminder of sparse linear solvers and multigrid methods, we introduce the underlying principles of *GmgPolar*. In Section 5, we then present the implementation of the solver, which is an adaptation of the already existing Matlab code from Martin Kühn into C++. The goal of our task is to improve the efficiency of the code, and to enable parallelisation for a future integration into the GyselaX code [37].

4 Geometric polar multigrid for a 2D Poisson-like equation with implicit extrapolation

In this section, we first describe the given two dimensional Poisson problem, before explaining how to construct it in generalised polar coordinates due to the disk-like domain. Next, we detail our approach, used in GmgPolar, for solving it with the help of finite difference discretization and multigrid. Given some general instruction on multigrid methods, we observe two major issues arising due to the polar coordinates: a singularity at the origin and an anisotropy of the grid. In this context, a special combined circle-radial zebra line smoothing procedure is developed. Furthermore, an implicit extrapolation technique is investigated in order to improve the accuracy of the approximated solution, as well as its suitability of being integrated into the multigrid scheme.

4.1 The given problem: a 2D quasi-neutrality Poisson-like equation

We focus on the solution of a simplified version of the two dimensional quasi-neutrality equation. This partial differential Poisson equation is defined on the cross-section $\Omega \subset \mathbb{R}^2$ of the tokamak as

$$\begin{aligned} -\nabla \cdot (\alpha \nabla u) &= f \qquad \textit{in } \Omega \\ u &= 0 \qquad \textit{on } \partial\Omega, \end{aligned} \tag{5}$$

see [39], where α is called the diffusivity coefficient, or density profile, and expressed as

$$\alpha(r) = \frac{2}{2.6 + 3.14}\left(1.3 + \arctan\left(\frac{1-r}{0.09}\right)\right). \tag{6}$$

The coefficient introduces an anisotropy due to the non-linear turbulence structures stretched along the magnetic field and models the rapid decay of the particle density from the core to the edge region in the tokamak cross-section, with a steep variation around 2/3 of the maximum radius [39], see Figure 10.

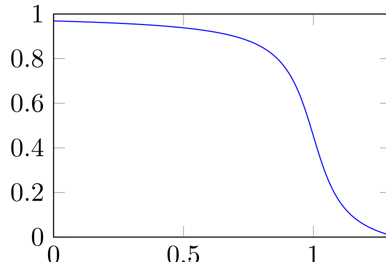


Figure 10: Variation of the diffusivity coefficient α over the radius r.

As the equation has to be solved on the disk-like cross-section, polar coordinates $(r,\ \theta)$ are more natural than Cartesian coordinates to represent this geometry. The actual physical domain $\Omega \subset \mathbb{R}^2$ can be described by a mapping F from a logical domain $\tilde{\Omega}:\ (0, 1.3) \times [0, 2\pi)$ as shown in Figure 11 [39].

In polar coordinates, the Poisson equation (5) reads

$$\begin{aligned} f &= -\nabla_{x(r,\theta),y(r,\theta)} \cdot [\alpha(r)\, \nabla_{x(r,\theta),y(r,\theta)} u(x(r,\theta), y(r,\theta))], \\ &= -\nabla_{x,y}\alpha(r)\, \nabla_{x,y} u(x,y) + \alpha(r)(-\nabla_{x,y} \cdot \nabla_{x,y} u(x,y)), \\ &= -\frac{\partial \alpha(r)}{\partial r}\, \frac{\partial u(x,y)}{\partial r} - \alpha(r)\, \Delta_{x,y} u(x,y), \end{aligned} \tag{7}$$

see [39], with the derivative of α (6)

$$\frac{\partial \alpha(r)}{\partial r} = -\frac{2}{(2,6 + 3.14)(0.09 + \frac{(1-r)^2}{0.09}}, \tag{8}$$

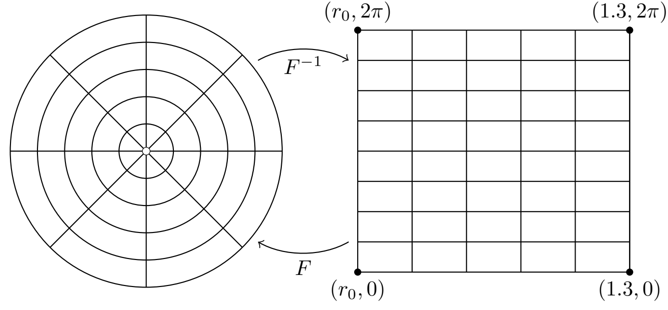


Figure 11: Mapping of the grid from Cartesian (Ω) to polar coordinates ($\tilde{\Omega}$), for the inverse mapping we have to exclude the origin ($r_0 > 0$) to avoid a singularity

and the Laplacian operator expressed in polar coordinates

$$\begin{aligned}\Delta_{x,y} u(x,y) &= \frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2}, \\ &= \frac{\partial^2 u(x,y)}{\partial r^2} + \frac{1}{r}\frac{\partial u(x,y)}{\partial r} + \frac{1}{r^2}\frac{\partial^2 u(x,y)}{\partial \theta^2}.\end{aligned} \tag{9}$$

Proof. Equation (9) can be proven as follows: Let's start by expressing function u with respect to r and θ as

$$\tilde{u}(r,\theta) = u(r\cos\theta, r\sin\theta) = u(x,y), \tag{10}$$

with the derivatives of the variables $x(r,\theta)$ and $y(r,\theta)$ with respect to r and θ being

$$\frac{\partial x(r,\theta)}{\partial r} = \cos\theta, \quad \frac{\partial y(r,\theta)}{\partial r} = \sin\theta, \quad \frac{\partial x(r,\theta)}{\partial \theta} = -r\sin\theta, \quad \frac{\partial y(r,\theta)}{\partial \theta} = r\cos\theta, \tag{11}$$

and the first and second derivatives of $u(x(r,\theta), y(r,\theta))$ with respect to r and θ

$$\begin{aligned}\frac{\partial u(x,y)}{\partial r} &= \frac{\partial u(x,y)}{\partial x}\cos\theta + \frac{\partial u(x,y)}{\partial y}\sin\theta, \\ \frac{\partial u(x,y)}{\partial \theta} &= \frac{\partial u(x,y)}{\partial x}(-r\sin\theta) + \frac{\partial u(x,y)}{\partial y} r\cos\theta, \\ \frac{\partial^2 u(x,y)}{\partial r^2} &= \frac{\partial^2 u(x,y)}{x^2}\cos^2\theta + 2\frac{\partial^2 u(x,y)}{\partial x \partial y}\sin\theta\cos\theta + \frac{\partial^2 u(x,y)}{\partial y^2}\sin^2\theta, \\ \frac{\partial^2 u(x,y)}{\partial \theta^2} &= r^2\left[\frac{\partial^2 u(x,y)}{\partial x^2}\sin^2\theta - 2\frac{\partial^2 u(x,y)}{\partial x \partial y}\sin\theta\cos\theta + \frac{\partial^2 u(x,y)}{\partial y^2}\cos^2\theta - \frac{1}{r}\frac{\partial u(x,y)}{\partial r}\right].\end{aligned} \tag{12}$$

Inserting these derivatives in (9) and with with $\sin^2\theta + \cos^2\theta = 1$, we get

$$\begin{aligned}\frac{\partial^2 u(x,y)}{\partial r^2} + \frac{1}{r}\frac{\partial u(x,y)}{\partial r} + \frac{1}{r^2}\frac{\partial^2 u(x,y)}{\partial \theta^2} \\ &= \frac{\partial^2 u(x,y)}{x^2}\cos^2\theta + \cancel{2\frac{\partial^2 u(x,y)}{\partial x \partial y}\sin\theta\cos\theta} + \frac{\partial^2 u(x,y)}{\partial y^2}\sin^2\theta, \\ &\quad + \cancel{\frac{1}{r}\frac{\partial u(x,y)}{\partial r}} \\ &\quad + \frac{\partial^2 u(x,y)}{\partial x^2}\sin^2\theta - \cancel{2\frac{\partial^2 u(x,y)}{\partial x \partial y}\sin\theta\cos\theta} + \frac{\partial^2 u(x,y)}{\partial y^2}\cos^2\theta - \cancel{\frac{1}{r}\frac{\partial u(x,y)}{\partial r}}, \\ &= \frac{\partial^2 u(x,y)}{x^2} + \frac{\partial^2 u(x,y)}{\partial y^2}\end{aligned} \tag{13}$$

and in fact obtain again the Laplacian operator in Cartesian coordinates, which completes the proof. □

In this work, three different physical domains are considered:

1. The simplest one, the circular geometry, see Figure 12a can be described by a standard polar coordinate transformation [38] defined as

$$F_1: \quad x = r\cos(\theta), \qquad y = r\sin(\theta). \tag{14}$$

2. In order to improve realism and to describe more realistic tokamak cross-sections [39], a mapping from polar coordinates to more realistic plasma shape is used [8]. The so-called Shafranov geometry, a stretched and shifted ellipse, see Figure 12b, is defined by the transformation

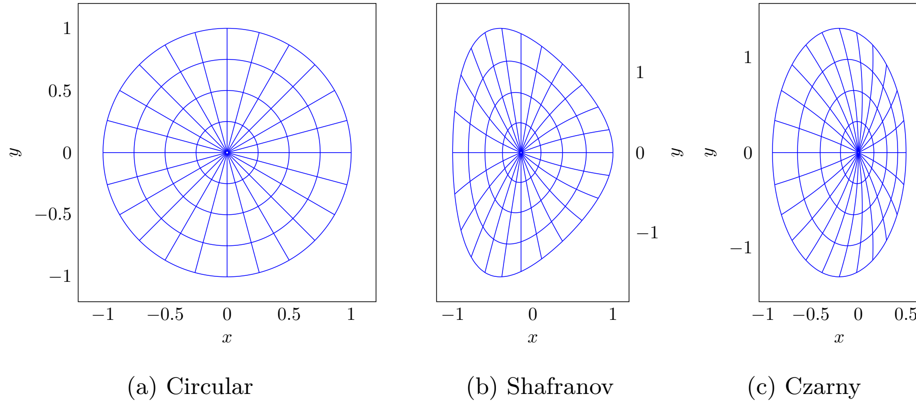
$$F_2: \quad x = (1-\kappa)r\cos(\theta) - \delta r^2, \qquad y = (1+\kappa)r\sin(\theta), \tag{15}$$

see [39], with the elongation $\kappa = 0.3$ and the Shafranov shift $\delta = 0.2$. For $\kappa = \delta = 0$, the Shafranov geometry reduces to a simple circular shape.

3. The third possibility, the Czarny geometry, see Figure 12c, is even more complex as it adds a triangularity to the shape

$$\begin{aligned} F_3: \quad x &= \frac{1}{\epsilon}\left(1 - \sqrt{1+\epsilon(\epsilon + 2r\cos\theta)}\right), \\ y &= y_0 + \frac{e\xi r\sin\theta}{2 - \sqrt{1+\epsilon(\epsilon+2r\cos\theta)}}, \end{aligned} \tag{16}$$

see [36], with $y_0 = 0$ being the mapping center, $\epsilon = 1.4$ the inverse aspect ratio, $e = 1.4$ the ellipticity and $\xi = 1/\sqrt{1-\epsilon^2/4}$.



(a) Circular (b) Shafranov (c) Czarny

Figure 12: Different geometries for the poloidal cross-section.

4.2 The finite difference discretization of the Poisson equation on disk-like domains

The original, elliptic boundary value problem, defined in the continuous domain Ω, may be discretized on a finite discrete grid [59, chapt. 3], e.g. using finite differences. In this case, the differential equation is replaced by a difference formula for every grid point [28, chapt. 2], with a finite number of *degrees of freedom* (DOFs) [59, chapt. 3]. We then obtain a discrete problem, which can be expressed in matrix form [10] as a sparse system of equations

$$Au = f, \tag{17}$$

see [64, chapt. 2], where A is of size $m \times m$.

In this section, based on [38], we introduce a linear system of type (17), obtained from a discretization of the PDE (5), which allows us to compute an approximated solution of the original

boundary value problem. For this purpose, we use a grid in the logical domain with respectively n_r and n_θ nodes in the r and θ directions.

In [59, chapt. 3], the authors state that a "discretization method should preserve symmetry[, in other words] when the PDE is symmetric, the coefficient matrix A should be symmetric as well". This property holds for example for a standard finite-difference discretization on uniform grids, which relates the discrete solution on a specific node to the continuous solution in the neighbouring grid points, which are together called a *stencil* [65, chapt. 3]. In the case of anisotropy or non-uniformity of the grid, a standard derivation of the finite difference discretization may lead to asymmetries. To maintain the symmetry of the stencil, the authors in [38] propose to consider the energy functional corresponding to the partial differential equation, and to derive a particular finite difference discretization using difference formulas to approximate the derivatives as well as elementary numerical integration rules.

The problem of finding a solution of the given partial differential equation (5) can be traced back to the energy minimizing problem

$$\min_u \; J(u), \qquad \text{with } J(u) = \int_\Omega \frac{1}{2} \nabla u^T \alpha \nabla u - fu \;\, d(x,y), \tag{18}$$

where $J(u)$ is the energy functional in the domain Ω. Using polar coordinates (r and θ), we can consider a rectangular mesh which is subjected to a curvilinear transformation, or mapping, F from Cartesian to generalised polar coordinates

$$F(\tilde{\Omega}(r,\theta)) = \Omega(x,y), \tag{19}$$

as introduced in the previous Section 4.1. Let's consider the Jacobian matrix DF of the mapping F, and its inverse DF^{-1} defined as

$$DF = \begin{pmatrix} \frac{\partial x(r,\theta)}{\partial r} & \frac{\partial x(r,\theta)}{\partial \theta} \\ \frac{\partial y(r,\theta)}{\partial r} & \frac{\partial y(r,\theta)}{\partial \theta} \end{pmatrix}, \quad \text{and} \quad DF^{-1} = \frac{1}{det(DF)} \begin{pmatrix} \frac{\partial y(r,\theta)}{\partial \theta} & -\frac{\partial x(r,\theta)}{\partial \theta} \\ -\frac{\partial y(r,\theta)}{\partial r} & \frac{\partial x(r,\theta)}{\partial r} \end{pmatrix}, \tag{20}$$

where $det(DF)$ is the determinant of DF. As done in [38] to simplify further notations, we also define $a^{rr}, a^{r\theta}, a^{\theta\theta} \in \mathbb{R}$ as

$$\frac{1}{2} DF^{-1} \alpha DF^{-T} \, |det(DF)| = \begin{pmatrix} a^{rr} & \frac{1}{2}a^{r\theta} \\ \frac{1}{2}a^{r\theta} & a^{\theta\theta} \end{pmatrix}. \tag{21}$$

Using equations (18), (20), and (21), we express the solution u within the energy functional $J(u)$ in polar coordinates, using the transformation F with

$$\begin{aligned} J(u) &= \int_\Omega \frac{1}{2} \nabla u^T \alpha \nabla u - fu \;\, d(x,y), \\ &= \int_{\tilde{\Omega}} \left(\begin{pmatrix} u_r^T \\ u_\theta^T \end{pmatrix}^T \frac{1}{2} DF^{-1} \alpha DF^{-T} \begin{pmatrix} u_r \\ u_\theta \end{pmatrix} - fu \right) |det(DF)| \;\, d(r,\theta), \\ &= \int_{\tilde{\Omega}} \begin{pmatrix} u_r^T \\ u_\theta^T \end{pmatrix}^T \begin{pmatrix} a^{rr} & \frac{1}{2}a^{r\theta} \\ \frac{1}{2}a^{r\theta} & a^{\theta\theta} \end{pmatrix} \begin{pmatrix} u_r \\ u_\theta \end{pmatrix} - fu \, |det(DF)| \;\, d(r,\theta). \end{aligned} \tag{22}$$

The global energy $J(u)$ can be express as a sum over the local energies $J_{R_{i,j}}(u)$ of all rectangular grid elements $R_{i,j}$ of the logical domain $\tilde{\Omega}$

$$J(u) = \sum_{i,j} J_{R_{i,j}}(u), \quad \text{with } J_{R_{i,j}}(u) = \int_{R_{i,j}} a^{rr} u_r^2 + a^{r\theta} u_r u_\theta + a^{\theta\theta} u_\theta^2 - fu \, |det(DF)| \;\, d(r,\theta), \tag{23}$$

with $0 \leq i < n_r - 1$ and $0 \leq j < n_\theta - 1$. As in [38], the local energy can now be approximated by applying the midpoint rule to the coordinates according to which the summands in the expression are differentiated. Subsequently, the partial derivatives can be approximated by a difference operator and the intermediate values can be replaced by an averaging operator using linear interpolation.

Finally, the remaining integral terms may be integrated using trapezoidal rule and we get a quadratic form for the approximated energy

$$\tilde{J}(u) = \sum_{i,j} \tilde{J}_{R_{i,j}}(u) \approx \sum_{i,j} J_{R_{i,j}}(u) = J(u), \tag{24}$$

where only function evaluations at the grid points $(r_i, \theta_j) \in [0, R] \times [0, 2\pi)$ remain. In order to minimize $\tilde{J}(u)$, we have to set its derivative with respect to $u_{s,t}$, with $(i,j) \in \{(s,t), (s-1,t), (s,t-1), (s-1,t-1)\}$,

$$\frac{\partial \tilde{J}}{\partial u} = \sum_{i,j} \frac{\partial \tilde{J}_{R_{i,j}}(u)}{\partial u_{s,t}} = 0 \tag{25}$$

to zero. Consequently, considering node $r(_i, \theta_j)$ with following interval size h_i and k_j, the expression in (25) has to be evaluated at the point itself as well as on the eight surrounding points, i.e. for every $i-1 \leq s \leq i+1$ and $j-1 \leq t \leq j+1$. This yields for all nodes, after reordering of the summands, a 9-point stencil. We here use the fixed letters a to i as short names for the position of the updates within the stencil.

$$\left] \begin{matrix} \textit{top left} & \textit{top} & \textit{top right} \\ \textit{left} & \textit{middle} & \textit{right} \\ \textit{bottom left} & \textit{bottom} & \textit{bottom right} \end{matrix} \right[\equiv \left] \begin{matrix} g & h & i \\ d & e & f \\ a & b & c \end{matrix} \right[, \tag{26}$$

The stencil updates are given in [26], depending on the coordinates and interval sizes of the current and neighbouring points,

$$\begin{aligned}
u_{i+1,j}: &\quad \textit{top} && := -\frac{1}{2} \frac{k_j + k_{j-1}}{h_i} (a^{rr}_{i,j} + a^{rr}_{i+1,j}), \\
u_{i-1,j}: &\quad \textit{bottom} && := -\frac{1}{2} \frac{k_j + k_{j-1}}{h_{i-1}} (a^{rr}_{i,j} + a^{rr}_{i-1,j}), \\
u_{i,j+1}: &\quad \textit{right} && := -\frac{1}{2} \frac{h_i + h_{i-1}}{k_j} (a^{\theta\theta}_{i,j} + a^{\theta\theta}_{i,j+1}), \\
u_{i,j-1}: &\quad \textit{left} && := -\frac{1}{2} \frac{h_i + h_{i-1}}{k_{j-1}} (a^{\theta\theta}_{i,j} + a^{\theta\theta}_{i,j-1}), \\
u_{i+1,j+1}: &\quad \textit{top right} && := -\frac{1}{4} (a^{r\theta}_{i,j+1} + a^{r\theta}_{i+1,j}), \\
u_{i+1,j-1}: &\quad \textit{top left} && := \frac{1}{4} (a^{r\theta}_{i,j-1} + a^{r\theta}_{i+1,j}), \\
u_{i-1,j+1}: &\quad \textit{bottom right} && := \frac{1}{4} (a^{r\theta}_{i,j+1} + a^{r\theta}_{i-1,j}), \\
u_{i-1,j-1}: &\quad \textit{bottom left} && := -\frac{1}{4} (a^{r\theta}_{i,j-1} + a^{r\theta}_{i-1,j}), \\
u_{i,j}: &\quad \textit{middle} && := -(\textit{top} + \textit{bottom} + \textit{left} + \textit{right}),
\end{aligned} \tag{27}$$

with the corresponding right hand side

$$\frac{(h_i + h_{i-1})(k_j + k_{j-1})}{4} f_{i,j} \, |det(DF_{i,j})|. \tag{28}$$

Note that in the case of a circular domain without any anisotropy, the 9-point stencil reduces to a 5-point stencil as all diagonal entries become zero. The entire derivation of the stencil can be found in [38].

As given in (5), the domain is restricted by Dirichlet boundary conditions on the outer radius $r_{max} = R$ [36]. When handling a Dirichlet boundary, the boundary conditions are incorporated into the system by eliminating the corresponding values [28, chapt. 2] from the system matrix A, and moving them to the right hand side f. In (29), the authors demonstrate how the incorporation of the boundary condition $u_\partial = g_\partial$ induces *ones* on the diagonal corresponding to the fixed values

on the boundary ∂ of a domain Ω. In order to conserve symmetry, we transform the system with A and f being both composed of the parts corresponding to the boundary and resp. the interior domain, as

$$\begin{pmatrix} A_{\partial\partial} & A_{\partial\Omega} \\ A_{\Omega\partial} & A_{\Omega\Omega} \end{pmatrix} \begin{pmatrix} u_\partial \\ u_\Omega \end{pmatrix} = \begin{pmatrix} f_\partial \\ f_\Omega \end{pmatrix} \quad \rightarrow \quad \begin{pmatrix} I & 0 \\ 0 & A_{\Omega\Omega} \end{pmatrix} \begin{pmatrix} u_\partial \\ u_\Omega \end{pmatrix} = \begin{pmatrix} g_\partial \\ f_\Omega - A_{\Omega\partial}\, g_\partial \end{pmatrix}. \tag{29}$$

The stencil for the Dirichlet boundary is therefore given as

$$stencil = \left]\begin{matrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{matrix}\right[. \tag{30}$$

Furthermore, periodic boundary conditions in θ-direction introduce the equivalence of the points at $\theta_0 = 0$ and $\theta_{max} = 2\pi$ [36]. In Section 4.4.2, we will discuss several different approaches to handle the singularity at the origin created by the use of generalised polar coordinates, including Dirichlet boundary conditions at $r_0 > 0$.

4.3 Introduction to Multigrid

We now want to solve the discretized problem $Au = f$ given in (17) with the help of multigrid methods, based on [59, chapt. 6]:

> “The multigrid iterative method is a powerful tool for the numerical solution of large sparse linear systems arising from the discretization [of partial differential equations]”.

Multigrid methods are especially used for elliptic PDEs but are also applicable for parabolic and hyperbolic PDEs, integral equations, evolution problems, etc. [10]. The characteristic feature of multigrid is its fast convergence [28, chapt. 1] and asymptotic optimal complexity [54]. Besides, it is one of the very few scalable algorithms and can be parallelised readily and efficiently [10]. According to [54], multigrid is not merely a single algorithm but rather a design principle for highly efficient solution algorithms. It is based on a recursion that cleverly combines different fine and coarse resolutions of the given differential equation, and a fast transport of information through the solution domain.

In this section, we remind the principles behind the solution of sparse linear systems of equations, and focus on the geometric multigrid method. For detailed introductions to multigrid methods, please refer to [64, 9, 28, 59, 65, 11].

4.3.1 Direct vs. iterative solvers

In order to solve the equation (17), i.e $Au = f$, multiple methods have been developed which are separated in two main categories: direct and iterative methods.

Direct methods are based on techniques for the elimination of variables in a matrix most commonly derived from the Gaussian elimination method [63]. In this method, a LU-factorization is performed on the matrix, such as $A = LU$, where L is lower triangular and U upper triangular. The factorization algorithm is given in Algorithm 1. Once the factorization has been performed, considering any right hand side vector f and defining a vector $y = Ux$, the equation $Ax = LUx = Ly = f$ can be solved for y by forward substitution, see Algorithm 2, and finally $Ux = y$ is solved for the solution x via backward substitution [63], see Algorithm 3. These substitution techniques are very simple and cheap, having a complexity in the order of $\mathcal{O}(2m^2)$. Thus, most of the cost is spent in the former computation of the factorization which computational complexity is of the order $\mathcal{O}(\frac{2}{3}m^3)$. For very large systems, this computational complexity may become forbiddingly expensive, and one has to turn to iterative methods.

Algorithm 1: Gaussian Elimination without pivoting

```
Input: U = A_{m×m}, L = I_{m×m}
for k=0:m-2 do
    for j=k+1:m-1 do
        l_{jk} = u_{jk}/u_{kk}
        for i=j:m-1 do
            u_{ji} = u_{ji} − l_{jk} · u_{ki}
        end
    end
end
Output: U, L
```

Algorithm 2: Forward Substitution

```
Input: L_{m×m}, b_{m×1}
for j=0:m-1 do
    y_j = (b_j − Σ_{k=0}^{j−1} y_k · l_{jk})/l_{jj}
end
Output: y
```

Algorithm 3: Backward Substitution

```
Input: U_{m×m}, y_{m×1}
for j=0:m-1 do
    x_j = (y_j − Σ_{k=j+1}^{m−1} x_k · u_{jk})/u_{jj}
end
Output: x
```

An iterative method, such as Jacobi or Gauss-Seidel, can be applied to (17). As an example, one iteration of the Gauss-Seidel scheme is given by

$$u^{i+1} = (D - A)^{-1}(U\, u^i + f) \tag{31}$$

see [28, chapt. 3], with $A = D - L - U$, D being the diagonal, $-L$ the lower triangular and $-U$ the upper triangular part of the matrix A.

When solving the discretized equation for one specific point, using the Gauss-Seidel scheme in contrast to the Jacobi iterative method, the new value for the current node is immediately updated by directly introducing it into the solution, and thus utilized when resolving for the next point [10].

However, depending on the problem, the Gauss-Seidel iteration itself can have a slow convergence [54]. In fact, iterative methods, such as Gauss-Seidel, are quite efficient for reducing high-frequency (oscillatory) components of the error, but the convergence is slow with respect to the lower frequencies (smooth modes) [28, chapt. 2]. In the context of multigrid, such solver is called a smoother (or relaxation method) and is said to satisfy the smoothing property, implying the convergence of the iterative method, since they are suitable to smooth the error of the approximated solution [65, chapt. 7]. This is the first fundamental idea of multigrid methods. The Gauss-Seidel smoother turns out to have a better, more remarkable smoothing effect than a damped-Jacobi smoother [64, chapt. 2], and is thus, primarily used in our algorithm *GmgPolar*.

After a fixed number of k relaxation steps, we get the approximation $u^{(k)}$ of the exact discretized solution u, and we can denote the error

$$e^{(k)} = u - u^{(k)} \tag{32}$$

and the residual

$$r^{(k)} = f - Au^{(k)}. \tag{33}$$

The residual equation

$$Ae = r \tag{34}$$

is equivalent to the original equation (17) since the approximated solution $u^{(k)}$ can be corrected by the error $e^{(k)}$ with

$$u = u^{(k)} + e^{(k)}. \tag{35}$$

4.3.2 The 2-level multigrid

In [43, chapt. 7], the authors state: "The idea of multigrid is that low frequency errors on a fine grid become high frequency errors on a sufficiently coarse grid", which is the second fundamental principle behind multigrid methods.

Let's consider that we have a coarser grid level on which the problem has been discretized additional to the fine grid. We denote h and H the interval sizes of the fine and coarse grids ($h > H$), and the corresponding linear problems and residual equations on both levels as

$$\begin{aligned} A_h u_h &= f_h, \qquad & A_h e_h &= r_h, \\ A_H u_H &= f_H, \qquad & A_H e_H &= r_H, \end{aligned} \tag{36}$$

where r_h and r_H are the current residual respectively on the fine and coarse grid. In order to approximately solve the residual equation on the coarser grid, an appropriate approximation of A_h on the coarser grid is required.

After smoothing the discretized equation on the fine grid, only low frequency errors remain [43, chapt. 7]. Since, a smooth error appears to be more oscillatory on a coarse grid than on a fine grid, these low frequency components can then be more effectively damped by relaxation of the residual equation on the coarser grid [10]. Consequently, the oscillatory part of the error is handled on the fine grid and the coarse grid is used to eliminate the smooth parts. After approximating the remaining error on the coarse grid, it can be used to correct the approximated solution on the fine grid using (35). This process is called the *coarse grid correction*, and is explained in [64, chapt. 3].

The remaining question is how to form the residual r_H on the coarse grid from the residual r_h on the fine grid, and how to use the approximated error from the coarse grid on the fine grid? To transfer data between the different grid levels, linear transfer operators are required [54]. The *restriction operator* R_h^H restricts the residual from the finer to the coarser grid, while the *prolongation operator* P_H^h interpolates the error in the opposite direction [64, chapt. 2]. For the prolongation operator the classical choice is bilinear interpolation, which is also well-defined for anisotropic meshes [39]. Considering a 2D-problem, the stencil depends on the coordinates of the current point, e.g. r and θ for our problem in sections 4.1 and 4.2, but for the isotropic case reduces to

$$stencil = \left]\begin{matrix} 1/4 & 1/2 & 1/4 \\ 1/2 & 1 & 1/2 \\ 1/4 & 1/2 & 1/4 \end{matrix}\right[, \tag{37}$$

see [64, chapt. 2]. Hereby, the coarse values are just mapped to the fine grid unchanged, corresponding to the middle of the stencil. The values at fine grid points not being coarse points are then computed as the averages of their direct neighbouring coarse points [10]. Depending on the position of the fine nodes to the coarse ones, either the two nodes left and right, or top and bottom are used with the weighting factor 1/2, or the four diagonal coarse nodes with the factor 1/4. For our discretization, as in [38, 39], the restriction operator is then simply defined as the transposed of the prolongation operator $R_h^H = {P_H^h}^T$ [11]. For other discretization, please note, that an additional weighting factor might be necessary to be integrated into the stencil, see [64, chapt. 2]. The operator on the coarse grid A_H can be defined either directly from a discretization of the PDE operator on the coarse grid, or using the Galerkin operator $A_H = R_h^H A_h P_H^h$.

Subsequently, one coarse grid correction step consists of the following: Computing the residual r_h on the fine grid h, followed by a fine-to-coarse transfer of information by restricting the residual to the coarse grid H, i.e. defining $r_H = {P_h^H}^T r_h$. Then, the residual equation is solved for the error e_H on the coarse grid, and subsequently, interpolated by coarse-to-fine transfer to finally correct the approximated solution with $u_h^* = u_h + P_H^h e_H$ on the fine grid.

However, taken on its own, the coarse grid correction process is not convergent [64, chapt. 2]. On account of its failure for oscillatory modes, the solution has to be smoothed before [10]. This implies that “it is necessary to combine the two processes of smoothing and [...] coarse grid correction” [64, chapt. 2]. As explained in [54], this combination, i.e. the alternating execution, leads to a very fast convergence, since both methods have complementary properties. The relaxations take care of finely resolved details, while the coarse grid correction transports information globally through the grid. Finally, we obtain our multigrid method to improve the approximated solution where at each iteration we apply successively a pre-smoothing, a coarse grid correction and a post-smoothing step [64, chapt. 2]. This scheme is called the 2-level multigrid, and can be further improved by the use of multiple coarser grids.

4.3.3 The multigrid V-cycle

In order to construct a hierarchy of N nested grids [39], the classical choice is standard coarsening [39]. Hereby, starting from a very fine grid, the mesh-size h is doubled in both directions [64, chapt. 2], so that the coarse grid is constructed by taking every other point from the original fine grid. Thus only about one fourth of the fine-grid points remain as coarse-grid points [59, chapt. 6] in a 2D problem. On each of these grid levels l of mesh-size h, the PDE equation is then discretized to obtain the operator A_l, and information is transferred between the grids by restriction and prolongation, which can be defined as bilinear interpolation again. The subscript l now refers to the level of the operator/vector instead of the size of the grid intervals. The 2-level multigrid can then be generalised using this hierarchy of linear systems defined on decreasing grids, whereby the number of total grids is given as N and the numbering starts with grid level $l = 0$ at the finest grid.

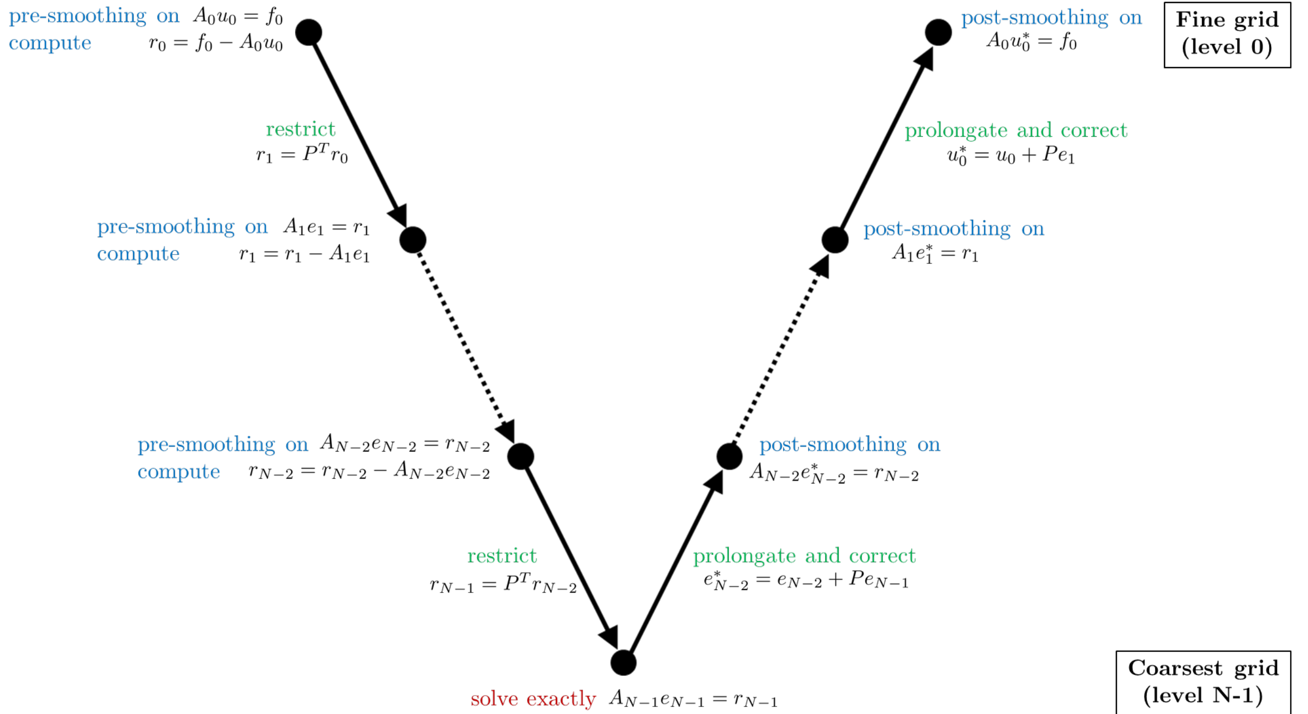


Figure 13: Schema of a multigrid V-cycle with N grids, denoting the finest level by 0. The final approximation of the solution after the V-cycle u^*_0 is more accurate than u_0 from the start.

The whole *multigrid cycle* starts with an arbitrary initial guess u_0 for the solution on the finest grid [44], followed by some relaxation sweeps on the initial discretized equation [9, chapt. 6]. After this step, the error will be sufficiently smooth and ready to be solved for on the next coarser grid [59, chapt. 6]. Consequently, the residual equation is transferred to the next level $l = 1$, to make

the error more oscillatory, and solved there by another subsequently smaller multigrid cycle [9, chapt. 6]. To be more precise, a few smoothing steps are performed on the residual equation on this grid level with the initial guess $e_1 = 0$ using the restricted residual r_1 as right hand side [10].

Successively, continuing this recursion using the given grid hierarchy and applying a smoother on all grids from the finest to the coarsest would, thus, annihilate all but the lowest frequency modes of the error on the coarsest grid [43, chapt. 7]. Furthermore, smoothing on a coarse grid is much cheaper than on a fine grid owing to the fact that less grid points need to be treated [10]. Assuming that the size of the problem on the coarsest grid is very small, the error at this level can then be eliminated by a direct solver [43, chapt. 7]. Ideally, the coarsest grid consists only of a few points [64, chapt. 2] and its order is so small that the cost of solving is negligible [59, chapt. 6]. Following, the approximated error on the coarse grids are transferred up the sequence of grids back to the finest grid as coarse grid corrections as in (35), combined with a few post-smoothing steps on each level [43, chapt. 7] "to smooth out any oscillatory error modes that may have contaminated the coarse-grid correction[s]" [59, chapt. 6].

The entire iterative procedure can be described schematically with the shape of the Latin letter '*V*', and is hence called a *V-cycle* [59, chapt. 6]. The whole multigrid V-cycle algorithm is illustrated in Figure 13 and detailed in algorithm 4.

Algorithm 4: The multigrid V-cycle on level l: $MG(l, u_l, f_l)$

while k < max_its iterations && no convergence attained **do**

1. Pre-smoothing:

   for v1 iterations **do**
   Smoothing on $A_l u_l^{(k)} = f_l$
   end

2. Coarse grid correction:

   - Compute the residual $r_l^{(k)} = f_l - A_l u_l^{(k)}$
   - Restrict the residual $r_{l+1}^{(k)} = P^T r_l^{(k)}$
   - **if** $l < N - 1$ (coarsest grid) **do**
     Call the multigrid cycle on level $l+1$: $MG(l+1, e_{l+1}^{(k)}, r_{l+1}^{(k)})$
     else
     Solve exactly on the coarsest grid: $e_{l+1}^{(k)} = A_{l+1}^{-1} r_{l+1}^{(k)}$
     end
   - Prolongate of the error $e_l^{(k)} = P e_{l+1}^{(k)}$
   - Correct of the solution $u_l^{(k^*)} = u_l^{(k)} + e_l^{(k)}$

3. Post-smoothing:

   for v2 iterations **do**
   Smoothing on $A_l u_l^{(k^*)} = f_l$
   end

- Set $u_l^{(k+1)} = u_l^{(k^*)}$

end

On a wide range of problem classes, multigrid methods display a mesh-independent convergence, and their complexity is of $O(m)$ operations, where m is the size of the system [43, chapt. 7] on the finest level. In order to solve the PDE problem introduced in sections 4.1 and 4.2, we use this V-cycle multigrid scheme, as defined in Algorithm 4, which poses several issues explained in the next Section . For more details about multigrid, consider [64, 65, 62, 43, 59, 9, 28, 10, 11, 49, 54, 44].

4.4 Two problems arising with the choice of polar coordinates

Multigrid methods for meshes in polar coordinates were already considered in, e.g., [3, 7, 44, 62, 64, 31] but are still less studied [39]. One of the reason are the two major problems arising with the coordinate transformation from Cartesian to generalised polar coordinates which treatment we detail in this section: the singularity at the origin and the anisotropy of the operator [3].

4.4.1 Singularity at the origin

The transformation from curvilinear to Cartesian coordinates introduces an artificial singularity at the origin for $r \to 0$ as the entire line $(r = 0 \times [0, 2\pi])$ is mapped to the single point $(x = 0, y = 0)$ and thus, multiple unknowns coincide geometrically. In [39], three approaches to overcome this problem are described.

The first workaround is to choose $r_0 > 0$ with, e.g., $r_0 \in \{10^{-2},\ 10^{-5},\ 10^{-8}\}$ and to simply enforce Dirichlet boundary conditions for the artificial boundary line at r_0. Obviously, these conditions are easily feasible for a manufactured solution, but hard (or even impossible) to determine in practical cases. Nevertheless, this approach is currently used in the GyselaX implementation as presented in [26].

Another method is to establish a discretization across the origin. To be more precise, we adapt the difference stencil by removing entries $a \equiv bottomleft$ and $c \equiv bottomright$, and changing entry $b \equiv bottom$

$$stencil = \left] \begin{matrix} g & h & i \\ d & e & f \\ 0 & \tilde{b} & 0 \end{matrix} \right[. \tag{38}$$

The new obtained entry

$$\tilde{b} = -\frac{1}{2}\,\frac{k_t + k_{t-1}}{2r_0}\,(a^{rr}_{0,t} + a^{rr}_{0,\tilde{t}}) \tag{39}$$

establishes a link between node (r_0, θ_j) and the node $\tilde{t}$ across the origin $(r_0, \theta_j \pm \Pi)$, $0 \leq j < n_\theta$ by considering the size of the interval $h_{-1} = 2\,r_0$, since the geometrical distance between these two nodes is twice the minimum radius r_0, and also by assuming that n_θ is odd, see [36].

A further, natural approach consists in integrating the node $(r_0, \theta) = (0, 0)$ into the mesh. Explicitly using the origin as discretization node, however, needs an adaption of the discretization of all nodes on $r_0 = 0$ using an average of the values corresponding to r_1 [3]. This workaround is studied in [39] where the authors find out that this discretization method has a negative impact on the convergence of the multigrid method, compared to the two previous approaches. Thus, we do not discuss this approach any further here.

4.4.2 Anisotropy

As a consequence of using polar coordinates, the mesh spacing at distinct radii is different [10], which naturally results in an anisotropic operator in the r-direction [3]. Furthermore, an additional refinement of the grid is required in order to capture the steep variation of α, see Figure 10, again leading to anisotropy in r [39]. Finally, the use of non-circular geometries, e.g. Shafranov or Czarny, creates an anisotropy also in the θ direction.

For highly anisotropic problems, point relaxation, such as Gauss-Seidel, in combination with standard coarsening does not yield satisfactory results [39] since the convergence factor highly increases [64, chapt. 5]. The reason for this is that point-wise relaxation has a strong smoothing effect only with respect to strongly-coupled degrees of freedom while it has poor smoothing properties with respect to weakly-coupled DOFs in the operator [64, chapt. 5]. In the context of multigrid, strong coupling between two points implies that the corresponding off-diagonal entry in the system matrix is relatively large compared to the other off-diagonal entries of the same DOF. On the contrary, we speak of weak coupling, if the entry is relatively small [39, 65].

The two classical workarounds to overcome anisotropy are semi-coarsening and line relaxation, which are explained in [64, chapt. 5]. In the case of semi-coarseining, a point-wise relaxation is kept, while the grid coarsening is adjusted. The coarse grid is, hence, defined by doubling the mesh size only in the direction of the strong-coupling, where the error appears smooth. However, since

more points remain on the coarse grid, this approach is not as fast as standard coarsening [10]. Additionally, we have anisotropy in both directions in our case, which also varies depending on the radius, so it would be quite complex to adapt the coarsening to the problem expressed in curvilinear coordinates.

The other possibility is changing from point-wise to line-wise relaxation while keeping standard coarsening. This means, the system is solved for entire lines of unknowns (constant r or θ) simultaneously. For each grid line i of unknowns, we define the sub-matrices of the matrix A:

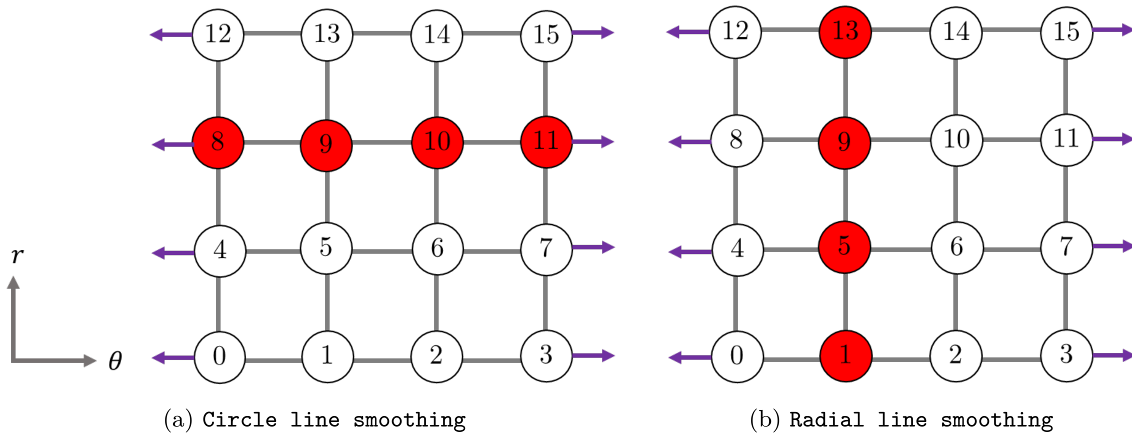
- A_i, corresponds to the stencil connections between nodes in same line i,
- f_i, the corresponding part of the right hand side,
- u_i, the corresponding part of the approximated solution,
- $A_i^{\perp}$, corresponds to the stencil connections between nodes in line i, and the nodes in neighbouring lines,
- $u_i^{\perp}$, corresponds to the part of the approximated solution of the neighbour lines.

Let's consider the use of block-Gauss-Seidel for our problem. Successively for each line, one must solve the linear system

$$A_i \cdot u_i = f_i - A_i^{\perp} \cdot u_i^{\perp} \tag{40}$$

for u_i, and update the corresponding part of the current approximated solution.

For the solution of our problem, we adapt the smoothing operation as explained, while keeping standard coarsening [39]. On a disk-like domain using polar coordinates, two natural line smoothing operations can be distinguished: the splitting of the grid in circular or radial lines [39]. Let's consider the example 4×4 grid shown in Figure 14, where the numbering of unknowns follows the θ-direction first.



(a) `Circle line smoothing` (b) `Radial line smoothing`

Figure 14: An example grid of size 4×4, using periodicity conditions in θ-direction. The indicated nodes are smoothed simultaneously, i.e. line relaxation in resp. circle and radial direction.

In the case of circle line smoothing all degrees of freedom of a circle with a constant r are relaxed together [36], see Figure 14a. For example, using the given stencil (26), and updating the unknowns u_8 to u_{11} (u_i) of the circle line collectively, gives the instance of (40) expressed as

$$\begin{pmatrix} e & f & & d \\ d & e & f & \\ & d & e & f \\ f & & d & e \end{pmatrix} \begin{pmatrix} u_8 \\ u_9 \\ u_{10} \\ u_{11} \end{pmatrix} = \begin{pmatrix} f_8 \\ f_9 \\ f_{10} \\ f_{11} \end{pmatrix} - \left(\begin{array}{cccc|cccc} b & c & & a & h & i & & g \\ a & b & c & & g & h & i & \\ & a & b & c & & g & h & i \\ c & & a & b & i & & g & h \end{array}\right) \left(\begin{array}{c} u_4 \\ u_5 \\ u_6 \\ u_7 \\ \hline u_{12} \\ u_{13} \\ u_{14} \\ u_{15} \end{array}\right). \tag{41}$$

The matrix A_i hereby takes a tridiagonal form with two additional elements in the upper right and the lower left corner due to the periodicity conditions. The connections of u_i to the unknowns from the previous (u_4 to u_7) and following line (u_{12} to u_{15}), i.e. $u_i^\perp$, are gathered in $A_i^\perp$ on the right hand side of the equation.

In the case of radial line smoothing, a line i corresponds to a constant θ [36], see Figure 14b. For instance, with the given stencil, the radial line composed of unknowns u_1, u_5, u_9, and u_{13} gives the instance of (40) expressed as

$$\begin{pmatrix} e & h & & \\ b & e & h & \\ & b & e & h \\ & & b & e \end{pmatrix} \begin{pmatrix} u_1 \\ u_5 \\ u_9 \\ u_{13} \end{pmatrix} = \begin{pmatrix} f_1 \\ f_5 \\ f_9 \\ f_{13} \end{pmatrix} - \left(\begin{array}{cccc|cccc} d & g & & & f & i & & \\ a & d & g & & c & f & i & \\ & a & d & g & & c & f & i \\ & & a & d & & & c & f \end{array}\right) \left(\begin{array}{c} u_0 \\ u_4 \\ u_8 \\ u_{12} \\ \hline u_2 \\ u_6 \\ u_{10} \\ u_{14} \end{array}\right). \tag{42}$$

In this case, the matrix A_i is tridiagonal, and the unknowns on the right hand side of the equation ($u_i^\perp$) are ordered according to the two neighbouring lines.

In [3] it was shown that both line smoothing operations behave very differently depending on the position within the circular domain [39]. In fact, circle relaxation shows better smoothing behaviour [39] around the origin than near the outer boundary, whereas radial smoothing is more efficient for $r \to r_{max}$ compared to the interior of the domain [3]. This is explained by the fact that the polar coordinates imply strong connections between degrees of freedom on circle lines [36] in a neighbourhood around the origin and, on the other hand, near the boundary, the strong connections lie in radial lines [3]. While updating strongly connected unknowns collectively works perfectly, resulting in smooth errors [39], line relaxation fails to achieve satisfactory convergence in the direction of weak coupling.

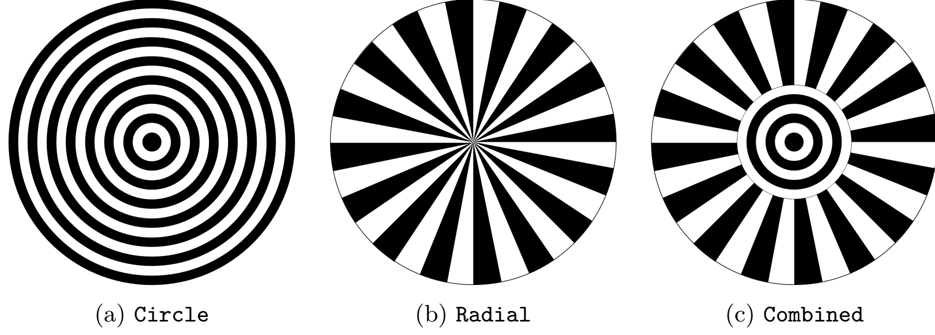
In [3], it was proposed to use a so called alternating-direction relaxation, where the whole domain is first treated by a circle smoother and subsequently by a radial smoother. In our case, in order to obtain a good convergence factor on the entire domain, we always update strongly coupled points together by a combination of the two line smoothing procedures on the domain, using circle relaxation near the origin and radial smoothing near the outer boundary. Thus, the domain is partitioned into two sections, where the two different smoothers are used [39]. In [39, 36], the authors show empirically that the best convergence is obtained when switching from circle to radial smoothing when

$$\frac{k_i}{h_j} r_j > 1. \tag{43}$$

In contrast to lexicographic line Gauss-Seidel smoothing, where the lines are handled one after the other, zebra line Gauss-Seidel smoothing consist of two half steps [64, chapt. 5], where all even and odd lines, respectively, are processed simultaneously [39]. Accordingly, all grid points with even line index are defined to be black and treated in the first half step while the remainder being white and handled in the second half step [65, chapt. 7]. In fact, the zebra line smoother has an improved convergence factor and a high degree of parallelism as all lines of the same colour are independent and thus, can be processed in parallel [64, chapt. 5]. Note that this only holds in case of using a stencil of length one which simply takes into account the direct neighbours. Otherwise, considering points from more lines for an update, additional colours have to be defined, and only the lines of one colour may be processed in parallel.

Figure 15 shows the colouring of a circular domain according to a circle, radial and combined circle-radial zebra smoother.

At each iteration of the combined smoother, we then alternate circle black, circle white, radial black, and radial white smoothers. Let's define for further notations the index *sc*, which indicates the *smoother* $s \in \{circle, radial\}$, and the *colour* $c \in \{white, black\}$, identifying the current smoother. For one sweep of zebra line relaxation all lines of the matrix belonging to the node indices of one smoother (circle or radial) and one colour (black or white) are then considered. We define the matrices and vectors A_{sc}, f_{sc}, u_{sc}, $A_{sc}^\perp$, and $u_{sc}^\perp$, as was done before but considering **all** lines corresponding to a smoother *sc*. Thus, using the definition (40), we have



(a) `Circle` (b) `Radial` (c) `Combined`

Figure 15: Different zebra line smoothers.

$$A_{sc} = \begin{pmatrix} A_{i_0} & & \\ & \ddots & \\ & & A_{i_k} \end{pmatrix}, \quad f_{sc} = \begin{pmatrix} f_{i_0} \\ \vdots \\ f_{i_k} \end{pmatrix}, \quad u_{sc} = \begin{pmatrix} u_{i_0} \\ \vdots \\ u_{i_k} \end{pmatrix}, \quad u_{sc}^{\perp} = \begin{pmatrix} u_{i_0}^{\perp} \\ \vdots \\ u_{i_k}^{\perp} \end{pmatrix}, \quad A_sc^{\perp} = \begin{pmatrix} A_{i_0}^{\perp} \\ \vdots \\ A_{i_k}^{\perp} \end{pmatrix}, \tag{44}$$

where $\mathbb{I}_{sc} = \{i_0, \ldots, i_k\}$ is the set of lines for a smoother *sc*. Consequently, there exist four different matrices A_{sc} and $A_{sc}^{\perp}$ for every combination of smoother and colour. As a consequence of using line relaxation, the collective solution of all equations corresponding to one smoother *sc* requires to solve a linear system of equations corresponding to each matrix A_{sc} in every smoothing step.

In this case, a LU-factorization by Gaussian elimination shall be applied to the matrix A_{sc} once during setup, followed by simple forward and backward substitution to solve the system in every iteration [36]. The decomposition of a matrix A into the lower triangular matrix L and the upper triangular one U can be performed using algorithm 1 [63]. Note that the number of operations required for the factorization is reduced drastically when considering the specific structures of the matrices corresponding to each smoother, as in (42) and (41).

Let us introduce another example grid of size 4×7 with Dirichlet boundary conditions on the inner radius and outer radius. Based on the criterion (43), the splitting between circle and radial smoothers is done between the third and fourth circle lines, see Figure 16.

Incorporating the Dirichlet boundary conditions into the system, as in (29), we get *ones* on the diagonal of the system matrix for all nodes on the boundary. Moreover, the connection between the circle and the radial smoothers needs to be considered for all radial lines, using the links to the circle points additionally in the vector $u_{sc}^{\perp}$. Therefore, except for the boundary, (41) still holds for the smoothing of one circle line, but (42) needs to be adapted for the radial line relaxation. For example, for the radial line of unknowns u_{13}, u_{17}, u_{21} and u_{24} in the 4×7 grid, the equations is reformulated as

$$\begin{pmatrix} e & h & & \\ b & e & h & \\ & b & e & \\ & & & 1 \end{pmatrix} \begin{pmatrix} u_{13} \\ u_{17} \\ u_{21} \\ u_{25} \end{pmatrix} = \begin{pmatrix} f_{13} \\ f_{17} \\ f_{21} \\ f_{25} \end{pmatrix} - \left(\begin{array}{ccc|cccc|cccc} a & b & c & d & g & & & f & i & & \\ & & & a & d & g & & c & f & i & \\ & & & & a & d & & & c & f & \\ & & & & & & 0 & & & & 0 \end{array}\right) \left(\begin{array}{c} u_8 \\ u_9 \\ u_{10} \\ \hline u_{12} \\ u_{16} \\ u_{20} \\ u_{24} \\ \hline u_{14} \\ u_{18} \\ u_{22} \\ u_{26} \end{array}\right). \tag{45}$$

The structure of the whole matrix A, based on the 4×7 grid from Figure 16, can be seen in Figure 17 with the separation between the unknowns of the two smoothers represented by a red line. Hereby, for the matrices A_{sc}, being the restrictions of A on the degrees of freedom of

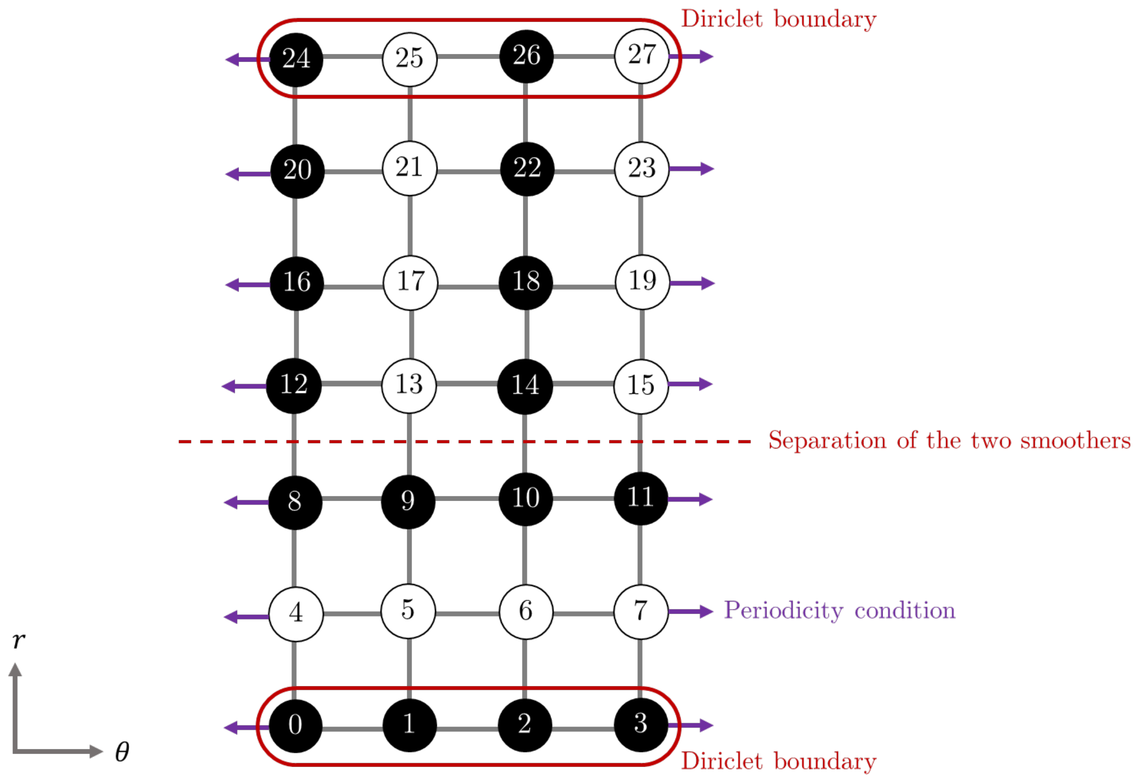


Figure 16: An example grid of size 4×7, using a combination of circle and radial smoother.

the corresponding smoother sc and indicating all connections to the same smoother, the color of each smoother (black or white) is shown explicitly. On the other hand, the complement matrices $A_{sc}^{\perp}$, corresponding to the remaining parts of the rows of A [22], consist of all non-coloured (grey) elements within the considered matrix lines.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
0	1																											
1		1																										
2			1																									
3				1																								
4					e	f		d	h	i		g																
5					d	e	f		g	h	i																	
6						d	e	f		g	h	i																
7					f		d	e	i		g	h																
8					b	c		a	e	f		d	h	i		g												
9					a	b	c		d	e	f		g	h	i													
10						a	b	c		d	e	f		g	h	i												
11					c		a	b	f		d	e	i		g	h												
12									b	c		a	e	f		d	h	i		g								
13									a	b	c		d	e	f		g	h	i									
14										a	b	c		d	e	f		g	h	i								
15									c		a	b	f		d	e	i		g	h								
16													b	c		a	e	f		d	h	i		g				
17													a	b	c		d	e	f		g	h	i					
18														a	b	c		d	e	f		g	h	i				
19													c		a	b	f		d	e	i		g	h				
20																	b	c		a	e	f		d				
21																	a	b	c		d	e	f					
22																		a	b	c		d	e	f				
23																	c		a	b	f		d	e				
24																									1			
25																										1		
26																											1	
27																												1

Figure 17: System matrix corresponding to the example grid of size 4×7: The red line indicates the separation between circle and radial smoother. Sections of the matrix corresponding to the four matrices A_{sc} are coloured in either black or white. The remaining elements correspond to the matrices $A_{sc}^{\perp}$.

4.5 Multigrid with implicit extrapolation

Implicit extrapolation is an efficient technique to improve the accuracy of a multilevel solver [34] by exploiting the information between grids on different discretization levels [6]. The basic idea is to use the extra information provided by multi leveling - through monitoring the change in the approximate solution from one level to the next - in order to give an estimate of the local error [56]. Eliminating certain dominating error terms, the convergence order of the approximated solution can be raised implicitly and inexpensively [52]. The approach might be suitable even for nonuniform [34], unstructured meshes [33] as well as for nonlinear problems [5] and could be easily integrated into a basic low order multigrid solver [34, 56].

In contrast to an explicit approach, i.e. the well-known Richardson extrapolation - where several discrete solutions on different levels are used directly in order to combine them linearly and to improve the error expansion - τ-extrapolation is more efficient [34, 55]. In fact, τ-extrapolation, which was originally introduced in [9, chapt. 8], is applicable in a more general case as it extrapolates the equation instead of the solution. In other words, the extrapolation is applied indirectly to intermediate quantities of the solution process [34] such as the truncation error, the residual, the energy, or the system matrix [6]. τ-extrapolation is particularly simple [45], because it avoids the explicit construction of higher order operators and complicated difference stencils [6]. It may be used implicitly [55] and can be applied in a natural form within the multigrid algorithm [34].

The τ-extrapolation is mathematically motivated by the asymptotic expansion of the truncation error [34] and based on the combination of such errors of several grid levels [55]. In general, the truncation error is defined as the correction to the right hand side that would make the solution u_h coincide with the true differential solution

$$A_h u_h = f_h + \tau^h, \tag{46}$$

see [9, chapt. 8]. With τ^H being the local truncation error on the coarse grid with mesh size H, and τ^h respectively on the fine grid with mesh size $h < H$, the following equality is valid up to a higher order in h

$$\tau^H \approx \tau^h + \tau_h^H, \tag{47}$$

see [9, chapt. 8]. The so-called relative truncation error $\tau_h^H := \tau^H - \tau^h$ is therefore defined as the difference of the coarse grid truncation error relative to the fine grid one. Being used to improve the approximated solution on the fine grid, τ_h^H is a correction term which would make the coarse grid error coincide with the fine grid error [9, chapt. 8].

In fact, higher order solutions can be obtained by appropriately combining systems of lower order equations [53] in such a way that the low order error terms cancel out [45]. Choosing suitable parameters [55] for a systematical linear combination [45], only higher order perturbations of the solution remain [56].

In [56], the energy extrapolation method in hierarchical basis formulation is discussed. Therein, the sequence of numerical methods $E_h(f)$ to approximate the energy of a function f

$$E(f) = \int_\Omega a\,(\nabla f(x,y))^2 d(x,y) \tag{48}$$

with a constant coefficient a has an h^2-expansion in the error

$$E_h(f) - E(f) = h^2 e_1 + h^4 e_2 + ... + h^{2N+2} R_{N+1}, \tag{49}$$

see [56] The remainder term R_{N+1} depends on the derivatives of f such that it vanishes if the derivatives of f of order $N+2$ vanish. When f is a quadratic polynomial, its derivatives of order $3 = N+2$ vanish and thus, (with N being 1) also $R_{N+1} = R_2$ does so:

$$E_h(f) - E(f) = h^2 e_1 + h^4 R_2 = h^2 e_1. \tag{50}$$

This means that the error expansion degenerates and that quadratics can be integrated correctly by using one single step of extrapolation

$$E(f) = 4/3 E_{h/2}(f) - 1/3 E_h(f), \tag{51}$$

see [56]. In the hierarchical basis representation higher order is simply obtained by multiplying certain matrix entries by the extrapolation factors 4/3 or 1/3 [56]. This is proven in [53] and can be quickly explained in the following way:

Consider a h^2 and a $(2h)^2$ solution and its error. Applying the following linear combination

$$4/3h^2 - 1/3(2h)^2 = 4/3h^2 - 1/3 \cdot 4h^2 = 0, \tag{52}$$

the h^2 error terms cancel out and, thus, the solution is of higher order than h^2 [57].

The theoretical basis for this is the global error expansion, whose existence depends on the smoothness of the solution [45]. Only if the solution is "sufficiently smooth, extrapolation leads to a convergence order which is higher than the approximation order of the basic discrete system" [5].

In our case, an implicit extrapolation method based on τ-extrapolation shall be employed. In the context of finite elements, a discretization based on linear elements is equivalent to a discretization with quadratic elements [45]. Typically, the convergence order is then raised from two to three by this extrapolation step [38] without explicitly calculating the solution on the coarser grid [55]. Moreover, in the experiments [39], we can see that an accuracy of even h^4 is reached, which can be explained by the canceling of all uneven order terms in the asymptotic h^2 error expansion in the case of uniform grid refinement [55].

In the multigrid context, we consider a coarse mesh, consisting of the set of coarse nodes denoted by the index c, and refine it by adding the intermediate points of all intervals to the set [38]. For an easier representation, a block partitioning of the matrix system is induced by ordering the nodes such that the coarse degrees of freedom appear first [34] and the intermediate fine ones, which are not in the set of coarse nodes and indicated by the index f, come second [38].

Consequently, two matrices A_C and A_F for resp. the coarse and the fine nodes can be assembled [38], where the fine matrix can be split according to the block partitioning

$$A_F = \begin{pmatrix} A_{F,cc} & A_{F,cf} \\ A_{F,fc} & A_{F,ff} \end{pmatrix}. \tag{53}$$

The extrapolated system is then a linear combination of both, with the extrapolated matrix A^{ex} and the corresponding right hand side f^{ex} given as [38, 33, 34, 53, 6]

$$\begin{aligned} A^{ex} &= \begin{pmatrix} 4/3A_{F,cc} - 1/3A_C & 4/3A_{F,cf} \\ 4/3A_{F,fc} & 4/3A_{F,ff} \end{pmatrix}, \\ f^{ex} &= \begin{pmatrix} 4/3f_{F,c} - 1/3f_C \\ 4/3f_{F,f} \end{pmatrix}. \end{aligned} \tag{54}$$

In the code, the extrapolated system is constructed without explicitly forming a block partitioning,

$$\begin{aligned} A^{ex} &= 4/3 \cdot A_F - 1/3 \cdot (P_{inj} \cdot A_C \cdot P_{inj}^T), \\ f^{ex} &= 4/3 \cdot f_F - 1/3 \cdot P_{inj} \cdot f_C, \end{aligned} \tag{55}$$

by the use of the transposed injection operator P_{inj}^T which maps the fine grid operators onto the coarse grid by simply taking the coarse nodal values unchanged while dropping all values on the fine nodes [53].

In [38], the following is proven for finite elements with nonstandard integration, and experimentally shown that it works accordingly for finite difference discretizations: When using two levels, a linear, nodal, non-hierarchical basis (A^l) can be transformed to an h-hierarchical basis (A^h) by a transformation T. The h-hierarchical can then be converted to a p-hierarchical basis (A^p) by implicit extrapolation (index ex) [34], which is then again equivalent up to the transformation T^T to a quadratic basis (A^q):

$$\begin{aligned} A^{l,ex} = T^T A^{h,ex} T = T^T A^p T = A^q \\ f^{l,ex} = T^T f^{h,ex} = T^T f^p = f^q. \end{aligned} \tag{56}$$

Figure 18 shows these four different bases.

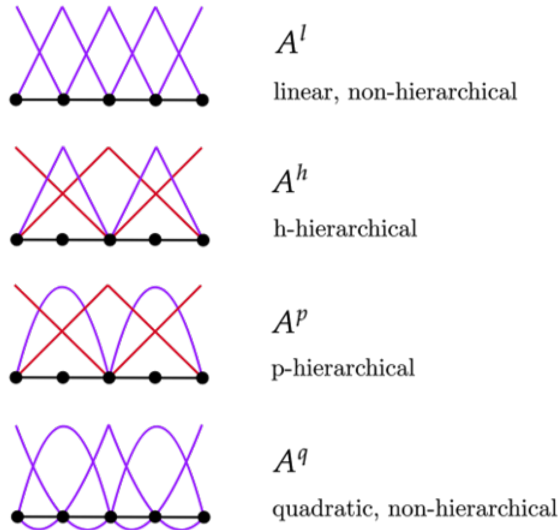


Figure 18: The four different bases: linear, h-hierarchical, p-hierarchical and quadratical, functions on the fine grid are marked in violet and functions on the coarse grid in red.

As the implicitly constructed, linear, extrapolated system matrix and respectively the right hand side coincide with the quadratically discretized operators, the following two systems yield the same solution [38, 33, 34, 6]:

$$\begin{aligned} A^{l,ex}u &= f^{l,ex}, \\ A^q u &= f^q. \end{aligned} \tag{57}$$

The convergence theorem is stated in [34]: The iterates of a multigrid algorithm using discretizations with piecewise linear functions and implicit extrapolation - by means of linear combination of the matrices of two different levels [56] - converge to the same solution with the same order of discretization error which we get by a discretization with piecewise quadratic functions. In summary,

> “the implicitly extrapolated multigrid algorithm for linear elements can be interpreted as a multigrid algorithm solving the original PDE when discretized by quadratic nodal basis functions” [39].

Since multigrid algorithms use a hierarchy of successively refined, nested grids [5], where the coarse levels are true subspaces of the finer levels [56], extrapolation is especially attractive to be included into the Algorithm [5]. Since we are not interested in the extrapolation of the error on the coarser grids but only on extrapolating the actual approximated solution, the “extrapolation step is only conducted between the two finest levels of multigrid hierarchy” [39]. In the interest of a successful extrapolation, the refinement between the two considered grids has to be uniform. Therefore, the mesh on the finest level $l = 0$ is constructed by uniform refinement from the second finest grid $l = 1$ by incorporating the midpoints of the intervals [38], resulting in an odd number of nodes in both directions.

As already mentioned, the grid points of the coarser mesh are denoted by c and the set of newly added fine nodes by f [56]. In a modified smoothing procedure all coarse nodes belonging to the set c are excluded and thus, only the fine nodes f not being coarse nodes are treated. The reason for this is that “a smoothing on the untransformed system is incompatible with the higher order accuracy of the extrapolated system” [56]. In order to avoid the explicit setup of the higher order system, we can just use the lower order equations for the smoothing step by neglecting the coarse nodes [57]. This special smoothing procedure [39] demarcates our implicit extrapolation from normal τ-extrapolation, where either the problem is ignored and thus all points are smoothed [56] or the relaxation on the finest grid is avoided at all after once having used extrapolation [64, chapt. 5].

Consequently, equation (40) with (44) for the smoothing procedure is reformulated for the finest level $l = 0$ as

$$A^{ex}_{sc,ff} \cdot u_{sc,f} = f^{ex}_{sc,f} - A^{ex,\perp}_{sc} \cdot u^{ex,\perp}_{sc}, \tag{58}$$

with sc denoting the smoother, as explained in Section 4.4.2. As the coarse nodes are excluded from the smoother sc, all elements of the corresponding nodes have to be shifted to the right hand side of the equation into the matrix $A^{ex,\perp}_{sc}$ (see example: fig. 19 and 20).

After the smoothing, which delivers the fine solution vector $u_{sc,f}$, the complete solution vector u_{sc} needs to be build from the coarse and fine solution:

$$u_{sc} = \begin{pmatrix} u_{sc,c} \\ u_{sc,f} \end{pmatrix} \tag{59}$$

Again, in the code, we do not use the reordering of the nodes as block partitioning form.

Subsequently, the residual of the extrapolated system needs to be calculated and restricted to the next level $l = 1$. Hereby, a modified restriction operator P^T_{ex} is used [39], which differs from general transposed bilinear interpolation due to the derivation of the stencil in the context of finite elements with the help of triangles. Therefore, the coarse nodes are just injected directly [39], while the fine nodes are examined whether they are connected to a coarse node by an edge of the coarse triangulation [56]. If there exists a direct link to a coarse node, the stencil entry for the fine nodes is 1/2 and else 0 [39]:

$$P_{ex} = \left]\begin{matrix} 1/2 & 1/2 & 0 \\ 1/2 & 1 & 1/2 \\ 0 & 1/2 & 1/2 \end{matrix}\right[. \tag{60}$$

As the refinement between the two finest grids is uniform, the stencil entries are always constant even in the case of anisotropy.

By applying P^T_{ex} to the residual of the extrapolated system on level 0 and inserting the given formulas (55) for the extrapolated matrix and right hand side (for $l = 0$, with $F = 0$ and $C = 1$), we get a detailed equation for the extrapolated residual restricted to level 1 [39, 33, 6, 34]:

$$\begin{aligned} r^{ex}_1 &= P^T_{ex}\, r^{ex}_0, \\ &= P^T_{ex}\,[f^{ex}_0 - A^{ex}_0 u_0], \\ &= P^T_{ex}\,[(4/3\, f_0 - 1/3\, P_{inj}\, f_1) - (4/3\, A_0 - 1/3\,(P_{inj}\, A_1\, P^T_{inj}))u_0], \\ &= 4/3\, P^T_{ex}\, f_0 - 1/3\, P^T_{ex} P_{inj}\, f_1 - 4/3\, P^T_{ex}\, A_0 u_0 + 1/3\, P^T_{ex} P_{inj}\, A_1\, P^T_{inj} u_0, \\ &= 4/3\, P^T_{ex}(f_0 - A_0 u_0) - 1/3\, P^T_{ex}(P_{inj}\, f_1 - P_{inj}\, A_1\, P^T_{inj} u_0). \end{aligned} \tag{61}$$

As $P^T_{ex} P_{inj} = I$ reduces to the identity and $f_0 - A_0 u_0 = r_0$ is the residual of the non-extrapolated system on the fine level $l = 0$, we finally get

$$r^{ex}_1 = 4/3\, P^T_{ex}\, r_0 - 1/3\,(f_1 - A_1\, P^T_{inj}\, u_0). \tag{62}$$

The multigrid algorithm with implicit extrapolation is given in algorithm 5. As explained, the extrapolation step is only conducted on the finest grid $l = 0$ and subsequently, a standard multigrid V-cycle (algorithm 4) is called on level $l = 1$ [39, 5].

Algorithm 5: The multigrid V-cycle on level $l = 0$ with implicit extrapolation: $MG^{ex}(u_0, f_0)$

while k < max_its iterations && no convergence attained **do**

1. Pre-smoothing:

    for v1 iterations **do**
    - Smoothing only on the fine nodes of $u_0^{(k)}$
    - Build the complete solution from the coarse and the fine solution

    end

2. Coarse grid correction:

    - Compute the residual $r_0^{ex,(k)} = f_0^{ex} - A_0^{ex} u_0^{(k)}$
    - Restrict the residual $r_1^{ex,(k)} = P_{ex}^T r_0^{ex,(k)}$
    - Call a standard multigrid cycle on level $l-1$: $\mathbf{MG}(l = 1, e_{l+1}^{(k)}, r_{l+1}^{ex,(k)})$ (Algo. 4)
    - Prolongation of the error $e_0^{(k)} = P_{ex} e_1^{(k)}$
    - Correction of the solution $u_0^{(k^*)} = u_0^{(k)} + e_0^{(k)}$

3. Post-smoothing:

    for v2 iterations **do**
    - Smoothing only on the fine nodes of $u_0^{(k^*)}$
    - Build the complete solution from the coarse and the fine solution

    end

- Set $u_0^{(k+1)} = u_l^{(k^*)}$

end

In Figure 19, the already presented example grid (compare fig. 16) of size 4×7 is adjusted by removing all the coarse nodes for the smoothing step on level 0. For the extrapolation to work, it is important to have an uneven number of grid points in both directions. Due to the periodicity condition in θ-direction the last line at $\theta = 2\pi$ which is equivalent to the first line at $\theta = 0$ is left out, resulting in Figure 19 in an even number of points again.

Figure 20 shows the structure of the resulting system matrix A^{ex} based on the grid from Figure 19. In fact, all lines corresponding to the coarse nodes have to be eliminated from the matrix and are hence, crossed out (in green) owing to the exclusion from the smoothing procedure. As already mentioned in this section, these coarse nodes need to be put to the right hand side of the equation into the matrix $A_{sc}^{ex,\perp}$. Consequently, the coloring (referring to the matrix A_{sc}^{ex}) of all elements within the corresponding columns (indicated by a red arrow) is disposed of, compared to Figure 16, in order to additionally incorporate the (red) encircled elements into the matrix $A_{sc}^{ex,\perp}$. Thus, a square form is achieved again for the matrix A_{sc}^{ex}.

To conclude, implicit extrapolation can be naturally combined with multigrid methods which then provides a very efficient solver for discrete systems [6] with an asymptotically optimal convergence [33]. It is competitive with multilevel methods using higher order directly [33] and therefore one of the most efficient approaches to the high accuracy solution of partial differential equations [6].

According to [39], a direct discretization with higher order, in contrast, typically leads to a dense matrix structure and thus, to a high flop cost per matrix-vector multiplication or smoother application. Using a clever recombination of low order components from the multigrid solver, an equivalent high order discretization can be constructed without the explicit setup of any more expensive, higher order, densely populated matrices, avoiding additional memory, memory traffic and high flop cost. Consequently, while resulting in a comparable convergence rate and numerical work per iteration, implicitly extrapolated multigrid has the advantage of a possibly simpler structure

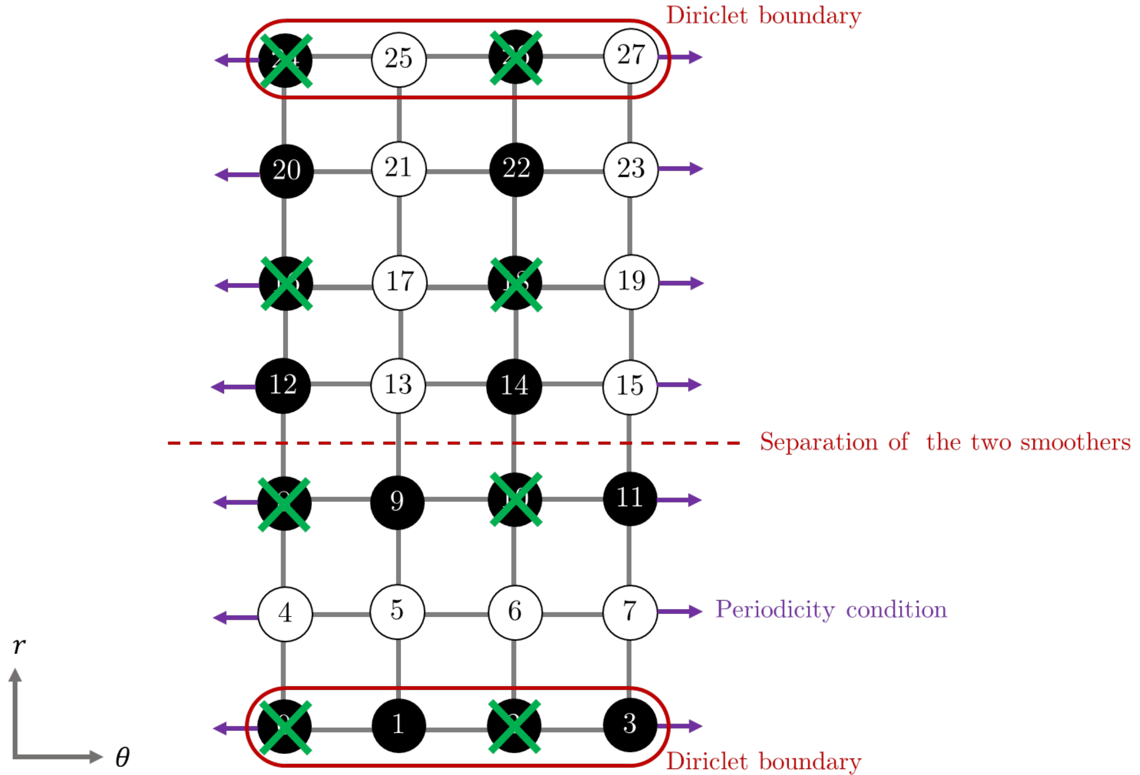


Figure 19: The already presented example grid of size 4×7 (compare to Figure 16) with the elimination of the coarse nodes for the smoothing procedure in case of implicit extrapolation. As a consequence, the node numbering changes in comparison to the previous example.

[33] and thus leads to a qualitatively equivalent high order discretization at reduced cost.

Compared to general multigrid without modification, the additional work for the extrapolation is very small [5]. In fact, except for the computation and restriction of the extrapolated residual, the cost is the same as for standard low order multigrid [39]. After all, implicitly extrapolated multigrid delivers higher order at only minimally raised cost [22].

On the other hand, as presented in [39], we can expect a slower algebraic convergence, meaning the iteration count is slightly larger. Furthermore, it is necessary to solve the discrete system up to a higher accuracy in order to exploit the lower discretization error. As a consequence of these two effects, the cost for the computation of a proper solution with extrapolated multigrid is still expected to be more expensive than with a general, basic, low order multigrid algorithm. Despite this fact, the iteration count still remains modest and independent of the mesh size, following that the solver is asymptotically optimal and scalable.

For any further literature about implicit extrapolation, please refer to [64, chapt. 5], [9, chapt. 8], [28, chapt. 14], others e.g. [55, 33, 34, 53, 5, 6, 56, 45].

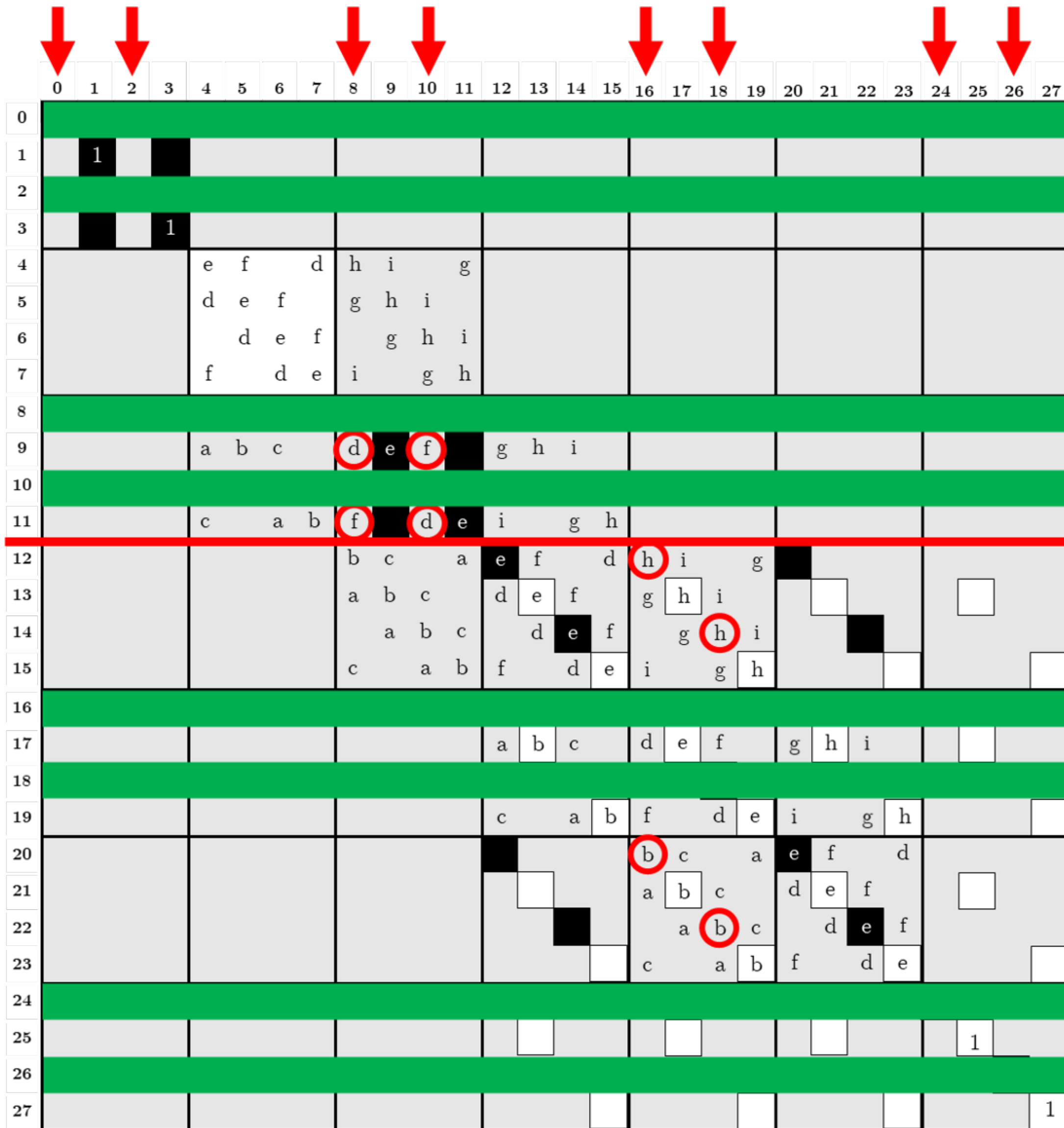


Figure 20: Extrapolated smoothing matrix corresponding to the example grid of size 4×7 in Figure 19. The elimination of the matrix rows corresponding to the coarse nodes is visualized with green lines. The red arrows indicate the matrix columns corresponding to the coarse DOFs, with the red circles showing all elements which have to be shifted from the matrix A_{sc}^{ex} to $A_{sc}^{ex,\perp}$.

5 Implementation of the solver GmgPolar

The multigrid algorithm for the gyrokinetic Poisson equation in polar coordinates, described in Section 3, with combined circle-radial zebra line smoothing and implicit extrapolation was implemented by Martin Kühn in Matlab within the project EoCoE. The next step is to convert the existing code to an object oriented C++ code in order to tackle larger problem sizes by improving efficiency and by enabling parallelisation as well as the simple integration of the solver into GyselaX. The purpose of the Matlab implementation was a simple development of the whole algorithm for research purposes, and thus contains also the comparison of several different options such as finite element and finite difference discretization, 5-, 7- and 9-point stencils, Jacobi and Gauss-Seidel smoothing, circle, radial and combined zebra line smoothing as well as different variants of extrapolation. Based on the results in [39] obtained with the Matlab implementation, we focus for the C++ implementation on an efficient, matrix-free implementation using a 9-point finite difference discretization with combined circle-radial zebra line Gauss-Seidel smoothing and the described implicit extrapolation method.

The implementation was done in the course of this master's thesis as a teamwork together with my supervisor Dr. Philippe Leleux, who provided a simple structure of classes and already implemented some of the basic functions concerning the setup of the problem, e.g. the construction of the right hand side f, and the application of the matrix A and the prolongation operator P. During this project my task was mainly to focus on the multigrid cycle with implicit extrapolation itself, while Dr. Leleux worked on the optimization of the functions.

In this Section, we first introduce the structure of the code, including the different parameters and object classes. Then we detail the implementation of the different elements required in the multigrid cycle, and finally, we demonstrate the efficiency of this new code in a series of numerical experiments.

5.1 Structure of the code

Our implementation has a very simple structure, shown in Figure 21. The code contains the two namespaces *param* and *gyro* as well as the two classes *gmgpolar* and *level*, which are all described in the following four subsections.

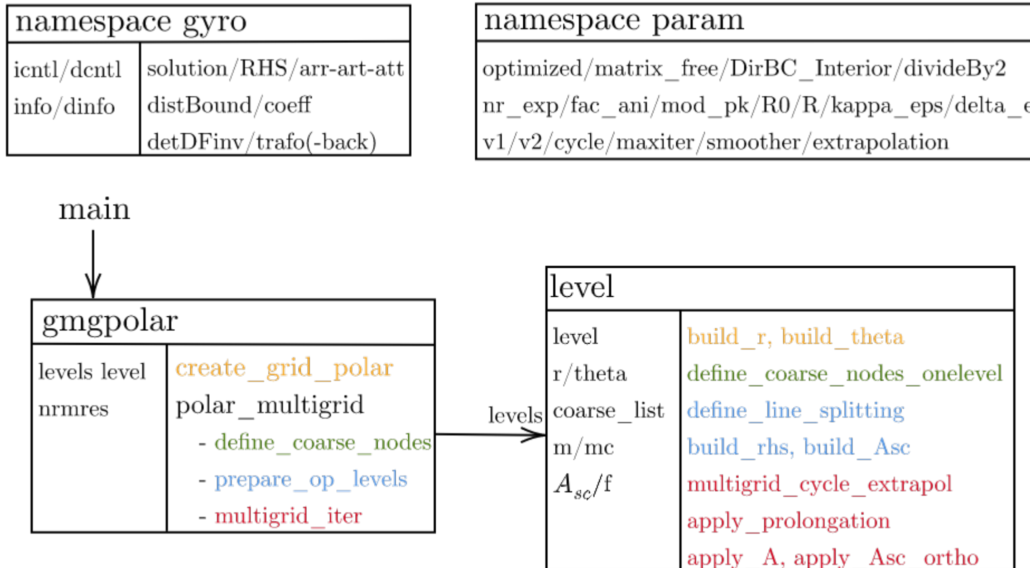


Figure 21: Structure of the GmgPolar C++ implementation. The class *gmgpolar* contains several instance of the class *level*, then the colors represent calls of methods in *level* from the methods in *gmgpolar*.

5.1.1 Namespace param

This namspace contains an enumeration for the definition of some global constants, listed in Table 1, which can be given to the program as parameters by the command line or are set to default values if not provided. The parameters define in particular how the grid is constructed, what geometry should be used, as well as what multigrid cycle is utilized.

By default, we use the optimised matrix-free code, with the Shafranov geometry ($\kappa = 0.3$, $\delta = 0.3$) on the ring with $r_0 = 10^{-5}$, $R = 1.3$, and a V-cycle multigrid with one pre- and post-smoothing step of the combined circle-radial smoother, as introduced in Section 4.4.2, and implicit extrapolation.

Type	Name	Description
code	*optimised*	Whether to use the optimised code version or not.
	matrix_free	Whether to use a matrix free implementation or not.
discretization	*nr_exp*	Defines the number of nodes in both direction: $n_r, n_\theta = \mathcal{O}(2^{nr_{exp}})$.
	fac_ani	Number of refinements around the steep jump of the coefficient α, i.e. around $r = 2/3R$.
	divideBy2	Defines how often to split the intervals of the grid at the midpoint.
	DirBC_Interior	Defines the treatment of the origin (0: across the origin discretization / 1: Dirichlet boundary).
geometry	*r0 / R*	Interior/ exterior radius of the disk-like shape.
	mod_pk	Defines the shape of the geometry (0: circular geometry / 1: Shafranov geometry / 2: Czarny geometry).
	kappa_eps	Parameter κ for the Shafranov geometry, parameter ϵ for the Czarny geometry.
	delta_e	Parameter δ for the target geometry, parameter e for the Czarny geometry.
V-cycle	*cycle*	Type of multigrid cycle (1: V-cycle / 2: W-cycle).
	maxiter	Maximum number of iterations.
	smoother	Defines the smoothing version (3: coupled circle-radial version/ 13: decoupled circle-radial version), see Section 6.2.
	v1 / v2	Number of pre- / post-smoothing steps.
	extrapolation	Defines the extrapolation version (0: no extrapolation / 1: implicit extrapolation / 2: implicit extrapolation with full grid smoothing, see Section 6.3).

Table 1: Parameters of the namespace *param*.

5.1.2 Namespace gyro

The namespace *gyro* contains the actual values of the parameters from *param* in the arrays *icntl* and *dcntl* as well as some fundamental functions for the computation of the right hand side, the exact solution, and the coefficients $\alpha, det(DF), a^{rr}, a^{r\theta}, a^{\theta\theta}$ from Section 4.2. Moreover, the namespace implements the mapping from Cartesian to polar coordinates. The most important functions are listed in Table 2.

Name	Parameters	Description
def_solution	(x, y) or $(r, theta)$	Returns the solution at the point (x,y) or (r,θ) or a vector of the solution on all grid points.
eval_def_rhs	$(r, theta)$	Returns a vector with the original right hand side f of the Poisson equation for all grid points.
coeff	r	Returns the diffusivity coefficient α depending on the radius.
detDFinv	$(r, theta)$	Returns the determinant of the derivative of F^{-1} at the point (r, θ), or a vector with the same for all points.
arr, *art*, *att*	$(r, theta)$	Returns the coefficient a^{rr}/ $a^{r\theta}$/ $a^{\theta\theta}$ at the point (r, θ), or a vector with the same for all points.
trafo, *trafo_back*	$(r, theta)$ and (x, y)	Transformation of the couple (r, θ), or of all points on the radius r, respectively from Cartesian to polar coordinates, and from polar to Cartesian coordinates.

Table 2: Functions from namespace *gyro*.

5.1.3 Class gmgpolar

The class *gmgpolar* stores a list of instances from the class *level* representing the multigrid hierarchy, numbered from the finest level $l = 0$ to the coarsest level $l = levels - 1$.

Name	Description
levels	Number of levels of the multigrid hierarchy.
v_level	Contains all grid levels of class level (1D array of instances of the class level).

Table 3: Variables from the class *gmgpolar*.

The functions defined in *gmgpolar* operate on all levels with a purpose such as the creation of the grid hierarchy, or the execution of the multigrid cycle, see Figure 21. In the tables 3 and 4, an overview over the most important variables and functions of the class is given.

Name	Parameters	Description
create_grid_polar	–	Creates the polar grids on all level.
create_grid_polar_divide	–	Refines the existing grid uniformly, by dividing at the center of the intervals.
polar_multigrid	–	Solves the linear problem with multigrid.
define_coarse_nodes	–	Defines the coarse nodes on all levels.
prepare_op_levels	–	Prepares the levels by constructing the operators.
multigrid_iter	–	Launches the overall multigrid cycle iterations.
multigrid_cycle_extrapol	l	Launches one multigrid cycle on level l.
compute_residual	l, *extrapol*	Computes the residual on level l, with or without extrapolation.
compute_error	–	Computes the error compared to the solution of the PDE equation.

Table 4: Functions from class *gmgpolar*.

5.1.4 Class level

This class represents one grid level of the multigrid hierarchy, identified by the number l, and thus, contains structures and variables that belong to one specific level, see Table 5. In particular, the class *level* stores, for instance, the solution u, the right hand side f and the residual res, which exist for every node on the 2-dimensional grid as one-dimensional vectors of length m in a row wise manner. Also, the matrices A_{sc} are stored in a sparse matrix format as three 1D arrays containing for each non-zero entry the row and column index, and the value.

Type	Name	Description
Grid	*l*	Index of the current level.
	nr, *ntheta*	Number of points in the r-, θ-direction.
	nr_int, *ntheta_int*	Number of intervals in r-, θ-direction.
	r, *theta*	List of coordinates in r-, θ-direction (1D-array).
	hplus, *thetaplus*	List of interval sizes in r-, θ-direction (1D-array).
Linear System	m, nz	Dimension and number of non-zeros of the matrix A.
	row_indices, *col_indices*, *vals*	Row / column / values of the nonzero elements in the sparse matrix A (1D-array).
	fVec, *u*, and *res*	Right hand side f, approximated solution, and residual vectors (1D-array).
	u_previous_c, *u_previous_r*	Approximated solution from the previous iteration of the circle / radial smoother (1D-array).
Coarser level	*coarse_nodes*	Number of coarse nodes.
	coarse_nodes_list_r, *_theta*	Indicating which nodes are fine (-1), and which are coarse (r and θ coordinates of the point) (1D-array).
	coarse_nodes_list_type	Indicating the position of coarse and fine nodes relative to each other (0: coarse node, 1: fine node with a coarse node on the same r-coordinate, 2-4: fine node with a coarse node on the same θ-coordinate, 5-8: fine node with diagonal links to coarse nodes) (2D-array).
	mc	Dimension of the matrix A on the next coarser level.
Smoother	*delete_circles*	Index of the last circular line belonging to the circle smoother.
	m_sc	Dimensions of the four matrices A_{sc} (1D-array).
	A_Zebra_r, *_c*, *_v*	Row / column / values of the nonzero elements in the four sparse matrices A_{sc} (2D-array).

Table 5: Variables from class *level*.

The class also defines functions which serve the construction of the grid on this level, the building and application of the operators A and P, as well as the smoothing procedure including the solving of the linear systems with the matrices A_{sc}, see Table 6. Most operators are not explicitly build and stored but applied directly to a vector in a matrix-free manner using a stencil representation. This is discussed in more detail in Section 5.2.1. For many of the functions, there exist an original naive implementation as well as an optimised version, see Section 6.1.

Type	Name	Parameters	Description
Grid	*build_r*, *_theta*	–	Builds the vector r, θ.
	define_coarse_nodes	finer_level	Defines the coarse nodes on one level.
Operators	*apply_A*	u, Au	Applies the matrix A to a vector u, mainly used for the residual computation.
	build_rhs	–	Builds the right hand side of the derived energy equation ($f \cdot u \cdot detDF$).
	apply_prolongation_bi, *_inj*, *_ex*	u, trans	Applies the prolongation operator P_{bi}, P_{inj}, P_{ex} for the bilinear interpolation. Option trans=1 apply the restriction (transposed prolongation). Returns the vector Pu.
Smoother	*multigrid_smoothing*	smoother	Smoothing procedure for one smoother *sc* solving for u_{sc} and inserting it into the overall solution u.
	define_line_splitting	–	Defines the switch circle to radial smoother.
	build_Asc	–	Builds the matrix A_{sc} for the smoother *sc*.
	build_fsc	f_sc, smoother	Builds the vector f_{sc} for the smoother *sc* from the vector fVec.
	apply_Asc_ortho	Au, u, smoother	Applies the matrix $A_{sc}^{\perp}$ for the smoother *sc* to a vector.
Linear System	*facto_gaussian*	A_row_indices, A_col_indices, A_vals, m_solution	Gaussian elimination to get the LU decomposition of a matrix A.
	solve_gaussian	A_row_indices, A_col_indices, A_vals, f	Forward-backward substitutions to solve a linear system of equations from the LU decomposition and right hand side vector f, return the solution vector.

Table 6: Functions from class *level*.

5.2 Implementation of the multigrid cycle

In this section, the implementation of the overall code is explained using the structure of classes introduced before. The whole code is divided into two steps: First, the setup phase constructs the grid hierarchy with its different levels, vectors and operators. Secondly, the multigrid cycle is called using elements constructed during setup.

In the main function, the following steps are carried out one after another:

1. call gmgpolar::create_grid_polar:
   - create the finest level (l=0)
   - call level::build_r and level::build_theta to create the vectors r and theta with the coordinates of the grid nodes
   - call gmgpolar::create_grid_polar_divide to refine the grid again
2. call gmgpolar::polar_multigrid:
   - call gmgpolar::define_coarse_nodes
     - call level::define_coarse_nodes_onelevel to create all coarser levels
   - call gmgpolar::prepare_op_levels
     - call level::build_A to create the operator on the coarsest level (l=N-1)
     - call level::build_rhs to build the right hand side for level 0 (and 1, in the case of extrapolation)

  - call level::facto_gaussian_elimination to factorize the operator A on the coarsest level (l=N-1)
  - call level::define_line_splitting to define the smoother and colour for every node
  - call level::build_Asc on all levels except for the coarsest
  - call level::facto_gaussian_elimination to factorize the operators Asc

3. call gmgpolar::multigrid_iter:
   - call gmgpolar::compute_residual and store the 2-norm of the residual vector on level 0
   - while (stopping criterium is not fullfilled) {
     - call gmgpolar::multigrid_cycle_extrapol on the finest level to perform one multigrid cycle
     - compute the convergence criterium

     }
   - call gmgpolar::compute_error to compute the 2- and inf-norm of the error vector

In the following subsections, we detail the contributions brought to each part of the code in the context of this master's thesis. First, we introduce the principle of matrix-free operations, essential for the performance of the solver, then we detail the multigrid solver split into the two steps: the setup phase, followed by the multigrid iterations with a specific focus on the implementation of the smoothing and the extrapolation scheme.

5.2.1 Matrix-free implementation

In the case where all operators are fully assembled and stored, a huge amount of memory would be required [36]. In order to overcome these memory issues, especially when considering large scale problems, a matrix-free implementation is of great importance, see [4] for more details. In such an implementation, the operators A, P and A_{sc} are not assembled (and stored in memory) but rather applied directly on vectors [36] using the stencil representation introduced in Section 4.2.

According to [36], the elements that we keep in memory for each level are the vectors r and $theta$ of size resp. n_r and n_θ defining the grid, as well as the right hand side f of size m, and the values for the coefficient α for every radius. In order to avoid the redundant computation of trigonometric functions, vectors with the values of $\cos\theta$ and $\sin\theta$ of size n_θ are stored as well. Additionally, the whole operator A on the coarsest grid ($\mathcal{O}(9m_c)$ floats), and the matrices A_{sc} ($\mathcal{O}(3m)$ floats) on every finer levels are required to be kept in memory in order to invert these matrices by applying a direct solver. In order to save memory, these matrices are stored in a sparse format, which means that only the subsequent nonzero elements of the matrix are stored in contiguous memory locations. However, this requires a scheme for knowing where the elements are positioned in the full matrix [18].

The simplest structure to store a sparse matrix is the coordinate list (COO), where the sparse matrix is represented as three one-dimensional arrays of size equal to the number of nonzero elements in the matrix [19, chapt. 2]. Hence, the nonzero values of the matrix are stored in a row-wise fashion in the floating point array *val*, while the corresponding column and row indices of the matrix elements are stored in the integer arrays *col* [18] and *row*. This results in a total storage of *#non_zeros* values [27], counting only the floats for the memory complexity in HPC. Another, even more efficient, way is the compressed row storage (CRS), where row pointers are stored instead of the row indices [27]. The row pointer array stores the location of the nonzero values in the array *val* that start a new row in the matrix [18]. Other possible sparse storage formats are for example the block CRS, compressed column storage (CCS) or compressed diagonal storage (CDS) [18]. The disadvantage of these formats is the difficulty of an indirect addressing step for every single value [18] as well as the insertion or deletion of matrix entries [27]. Therefore, at this moment, the simple COO format is used in order to keep matrix vector operations as simple as possible while only storing the nonzero values.

In conclusion, as derived in [36], the memory cost on a level l is of the order $O(4m)$ (f and A_{sc}), where m is the total number of the grid points. Neglecting the storage of the coarsest level

operator and considering that the number of points is divided by approximately 4 when going to a coarser level, we get the total cost in memory

$$Mem = \sum_{l=0}^{levels} \mathcal{O}(4\frac{m}{4^l}) \xrightarrow[L\to\infty]{} O(5.33m). \tag{63}$$

The complexity of the whole solver can be considered as linear with $O(m \cdot (C_{setup} + it \cdot C_{it}))$ flops, where $m \cdot C_{setup}$ and $m \cdot C_{it}$ are the costs resp. for the setup and per iteration. In [36], the authors show that the final implementation of GmgPolar has $C_{setup} = 56$ and $C_{it} = 268$.

5.2.2 Problem setup

The setup of the solver is only done once at the beginning of the code and then used in every iteration of the multigrid cycle. It consists of several steps for each level: the construction of the grid, the definition of the next coarse grid, and the construction of the smoother matrices A_{sc}.

Construction of the grid

The construction of the fine grid is performed by the functions *level::build_r* and *level::build_theta*, first defining the coordinates of the nodes in r-direction and subsequently in θ-direction. The parameter *fac_ani* allows to switch on additional refinements around the radius where there exists a steep variation of the coefficient α, i.e. $r_\alpha = 2/3\,r$, resulting in a larger number of total nodes and creating an anisotropy in r-direction:

- Without anisotropy, a total number of $n_r = 2^{nr_exp-1}$ equally distributed nodes in the r-direction is obtained.
- However, when anisotropy is established, the grid is additional refined around approximately r_α . First, $n_r = 2^{nr_exp} - 2^{fac_ani}$ equally distributed nodes are created and next, the grid is refined *fac_ani* times at the center of the intervals in the region around r_α. For instance, with *nr_exp=4* and *fac_ani=3*, we get $nr = 10$ equally distributed nodes, and then recursively split the intervals around r_α in the middle three times, resulting in a total number of 25 nodes in the r-direction, see Figure 22.

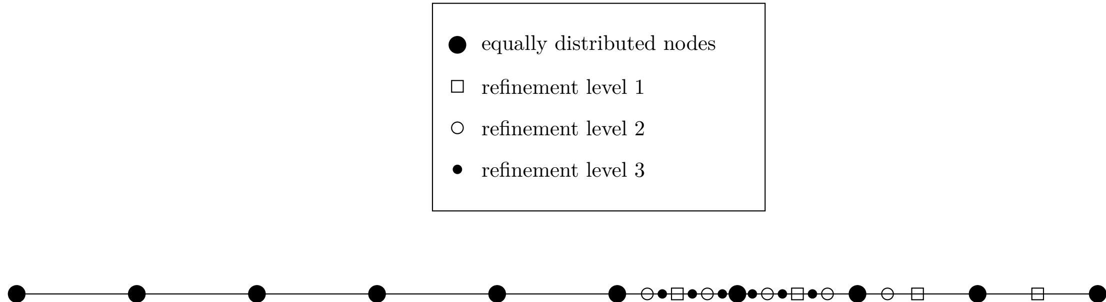


Figure 22: Anisotropic refinement of the grid around $r_\alpha = 2/3R$ in the r-direction.

Lastly, regardless of using any anisotropy or not, the whole grid is refined again uniformly in the r-direction by splitting at the center of all intervals in order to enable a uniform grid refinement between the two finest levels for a successful extrapolation, as defined in Section 4.5. In the example case with $n_r = 25$, we thus get a total number of $nr = 2 \cdot 25 - 1 = 49$ nodes in the r-direction.

After the creation of the coordinates in r-direction, the number of points in θ-direction is calculated from n_r via $n_\theta = 2^{ceil(log_2(n_r))}$. For $n_r = 49$, the total number of nodes in θ-direction is, hence, $n_\theta = 2^{ceil(log_2(49))=2^6=64}$. We define a uniform division for the θ-direction, i.e., all coordinates are distributed equally over the angle of 2π. Hereby, the first θ-coordinate ($\theta = 0$) coincides with the last one ($\theta = 2\pi$).

Finally, in the case of having a parameter *divide_by2* greater than zero, the grid is again refined uniformly in both directions by splitting all intervals at their center points *divide_by2* times by

calling the function *level::create_grid_polar_divide*.

Coarse nodes and prolongation
For the application of the prolongation operator of dimension $mc \times m$, four different types of nodes need to be considered with the specified stencil [36]. In Figure 23, the empty circles correspond to the coarse nodes and the plain circles to the fine ones respectively [22]. As the prolongation of a fine node depends on its position in the grid with respect to the neighbouring coarse nodes [22], there are three different types of fine nodes to distinguish: nodes with neighbouring coarse nodes resp. in the radial or polar direction, and nodes with neighbour coarse nodes only in the diagonals [22].

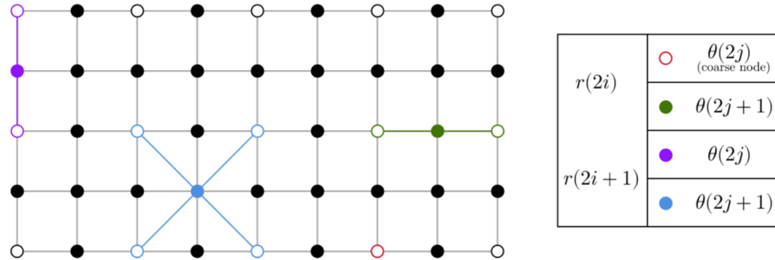


Figure 23: Stencil representation for 4 different types of nodes, depending on their position with respect to the neighbouring coarse nodes [36].

In the code, the 2-dimensional array *coarse_node_list* of class *level* is used to store these types of nodes. Hereby, the indices of the nodes on the fine grid are stored in the 1D array which corresponds to the type of the node within the 2D array. Additionally, the indices of the neighbouring coarse nodes on the coarse grid are stored for every fine node in the same array, which allows for a fast application of P. In order to distinguish the index of the fine node from the ones of the neighbour nodes, the variables *shift* and *start* indicate the number of values in the list per node as well as the position of the fine index itself within these values.

Matrices A_{sc} for the smoothing of the grid
In order to build the matrices A_{sc} explicitly on all levels at the beginning of the code, the function *build_Asc* is called for all levels l except the coarsest. This function creates the four matrices A_{sc}, as defined in Section 4.4.2, and stores them in COO sparse format in the 2D arrays *A_Zebra_r*, *A_Zebra_c* and *A_Zebra_v*. The four matrices are stored in the following order:

0. circle black smoother
1. circle white smoother
2. radial black smoother
3. radial white smoother,

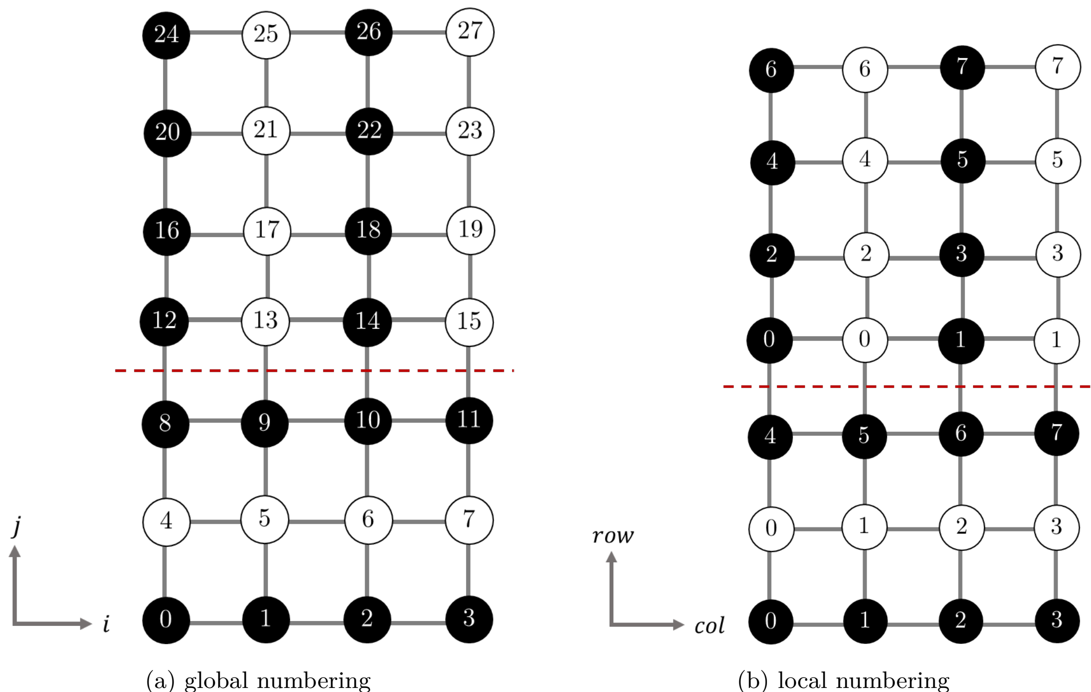
with their size stored in the array *m_sc*[4]. For example, *A_Zebra_r[0]*, *A_Zebra_c[0]*, and *A_Zebra_v[0]* stores the matrix $A_{CircleBlack}$.

Within the function *build_Asc*, we iterate over all grid points, using the indices $i \in \{0, \ldots, n_\theta\}$ and $j \in \{0, \ldots, n_r\}$ resp. for θ and r, which is different to the indexing described in 4.2. Identifying the corresponding *smoother* for every node (0 to 3) as defined in Table 7, every point is then treated accordingly to this *smoother*.

First, we define a local numbering for each smoother, again following points first in the θ-direction, see Figure 24 for an example of global and local numbering in a 4×7 grid. Thus, the local coordinates (row, col) of the node within its smoother have to be computed from the global ones (i, j), as less points are used for the matrices A_{sc} than for the whole matrix A. For the circle smoothers, the coordinate in θ-direction (col) stays the same while the coordinate in r-direction (row) is divided by two. For the radial smoothers, on the other hand, col is divided by two and row has to be reduced by the number of lines in the circle smoother. Table 8 summarizes this mapping between local and global coordinates for all smoothers.

Smoother	Black	White
Circle	$j < delete_circles$	
	even index j	odd index j
Radial	$j \geq delete_circles$	
	even index i	odd index i

Table 7: Identification of the smoother for a node i/j.



(a) global numbering

(b) local numbering

Figure 24: Example 4×7 grid with both global numbering and numbering local to each smoother.

Smoother	*col*	*row*
Circle	i	$floor(j/2)$
Radial	$floor(i/2)$	$j - delete_circles$

Table 8: Mapping from global coordinates (i, j) in the grid, to local coordinates (row, col) for all smoothers.

As (row, col) only represent the coordinates of the node in the grid, the index within the matrix A_{sc} needs to be computed subsequently:

$$\begin{aligned} \text{circle smoother:} \quad & index = row \cdot ntheta + col \\ \text{radial smoother:} \quad & index = row \cdot n_cols_r + col, \end{aligned} \tag{64}$$

where $ntheta$ is the number of points in the θ-direction, and $n_cols_r = ntheta_int/2$ is the number of radial lines in the grid for the two radial smoothers. Another approach to obtain this *index*, which was used later throughout optimizations, is to keep count of all previously treated nodes as we iterate over them. We store the current number of nodes separately for the four smoothers in the array *count_nodes[4]*, and then use the current amount of nodes as *index*.

Subsequently, we need to go through the 9 points of the stencil for every node. Figure 25 shows the decomposition of the polar plane into circle and radial smoothers with the black/white coloring [22]. Additionally, the stencils for the matrices A_{sc} and $A_{sc}^{\perp}$ are displayed together with the function evaluations a^{rr}, a^{rt} and a^{tt}, which are given to the stencil for each update [22]. We

can observe that the matrix A_{sc} only uses the middle as well as the right/left updates in case of the circle smoother (resp. top/bottom for the radial smoother). On the other hand, the matrix $A_{sc}^{\perp}$ employs the diagonal updates along with the top/bottom updates for the circle smoother (resp. right/left for the radial smoother).

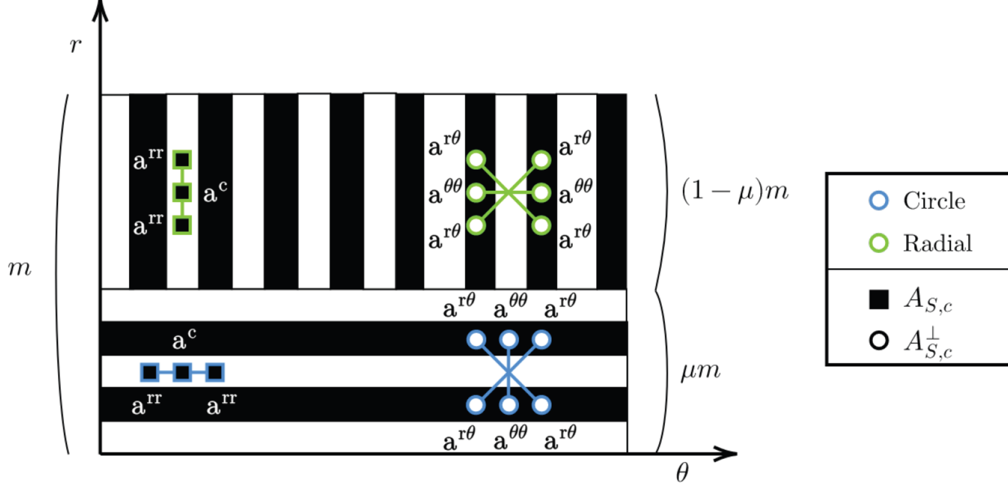


Figure 25: Stencils of the matrices A_{sc} and $A_{sc}^{\perp}$ for each smoother in the polar plane [22].

Having computed the index of each point, i.e. their row index within the matrix, with (64), we still need to compute the column index c for each nonzero entry. This index is defined by the position in the 9-point stencil, indicating whether it corresponds to a link to itself or to the top, bottom, right or left neighbour. Table 9 shows which links are treated depending on the smoother, and how the column index c is computed for the different updates of the stencil.

Exceptions	PB: add nt to c		PB: subtract nt from c
not for the last row	~~**top left**~~	**top:** $c = index + n_cols_r$ only for radial smoothers	~~**top right**~~
	left: $c = index - 1$ only for the circle smoothers	**middle:** $c = index$ for all smoothers	**right:** $c = index + 1$ only for the circle smoothers
not for the first radial line	~~**bottom left**~~	**bottom:** $c = index - n_cols_r$ only for the radial smoothers (AOD: first line of the circle smoother)	~~**bottom right**~~

Table 9: Application of the stencil to build the matrix A_{sc}. *PB*: indicates that a node lies on the periodic boundary. *AOD*: denotes the across-the-origin discretization. $nt = ntheta_int$.

Obviously, no top update is required for the last row ($r_j = r_{max} = nr - 1$) and no bottom update for the first radial row ($r_j = delete_circles$) as the link from the radial to the circle smoother is part of the complement matrix $A_{sc}^{\perp}$. Furthermore, for any point on the periodic boundary, the number of nodes per line ($ntheta_int$) have to be either added or subtracted depending on the left or right side of the stencil. In the case of an across-the-origin discretization, the first row of the circle smoother has an additional bottom update, as the nodes are linked in this direction with points from the same line and thus of the same smoother.

Lastly, the value of the matrix entry is computed according to the stencil using the global functions from the class *gyro* for the computation of the coefficients α, a_{rr}, $a_{r\theta}$, $a_{\theta\theta}$ and $det(DF^{-1})$.

Finally, this entry is added to the sparse representation of A_{sc} by including *index* into A_Zebra_r, c into A_Zebra_c and the value into A_Zebra_v.

5.2.3 The multigrid cycle itself

The multigrid cycle starts with a call to the method *gmgpolar::multigrid_cycle_extrapol* on level $l = 0$, which then recursively calls itself again on level $l+1$ - and thus again and again - down to the coarsest grid. The structure of the function *multigrid_cycle_extrapol* is presented in the following:

1. Pre-smoothing:
    - call *level::multigrid_smoothing* for all four smoothers
2. Coarse-grid-correction:
    - call *gmgpolar::compute_residual*
    - if (extrapol=1 and level=0){
        - restrict the extrapolated residual by calling *level::apply_prolongation_ex* and *level::apply_prolongation_inj* (with trans=1) , as well as *level::apply_A*

      } else {
        - restrict the residual by calling *level::apply_prolongation_bi* (with trans=1)

      }
    - if (second coarsest level) {
        - call *level::solve_gaussian_elimination* to compute the coarse error on the coarsest grid

      } else {
        - recursively call *gmgpolar::multigrid_cycle_extrapol* to do a multigrid cycle on the next coarser level

      }
    - if (extrapol=1 and level=0) {
        - prolongate the extrapolated residual by calling *level::apply_prolongation_ex* (with trans=0)

      } else {
        - prolongate the residual by calling *level::apply_prolongation_bi* (with trans=0)

      }
    - correct the solution by adding the fine error to the current solution u
3. Post-smoothing:
    - call *level::multigrid_smoothing* for all four smoothers

A particularly important aspect about the code, which has to be mentioned, is that the error at level l becomes u on level $l + 1$, while the residual on the fine level becomes the right hand side f on the coarser level, according to the principle of the coarse grid correction. Therefore, after the computation of the residual on level l, it is restricted to the coarser level and stored there in *fVec*. The solution u, corresponding to the error on the finer level, needs to be reset to zero in every iteration on each level except the finest, where it in fact represents the approximated solution.

After the smoothing and the residual restriction, we need to check which level we are currently on, before recursively calling the function *gmgpolar::multigrid_cycle_extrapol* again on level $l + 1$. When reaching the second coarsest level ($l == levels - 2$), the system on the next level (coarsest level) has to be solved for the error with a direct solver. The resulting error is then prolongated as usual and used as correction to the current solution u on the next finer level. Finally, the post-smoothing is conducted exactly in the same way than the pre-smoothing procedure.

The whole multigrid cycle is called iteratively in a while-loop until the convergence criterion is fullfilled, or the maximum number of iterations is reached. To monitor the convergence of the solution we use the relative residual which is computed by dividing the 2-norm of the current residual by the 2-norm of the initial residual.

Smoothing

The function *level::multigrid_smoothing* is called four times to execute the smoothing procedure for each smoother (0 to 3) within one smoothing step.

As already mentioned in the previous section, the matrix A_{sc} is of smaller size compared to the whole matrix A and therefore, also the vectors u_{sc} and f_{sc} are only subvectors resp. of u and f, whose size is stored in the array $m_sc[4]$. For example, for a grid of size $n_r \times n_\theta = 49 \times 64 = 3136$ with 11 lines belonging to the circle smoother, the number of points of the different smoothers are $m_sc[0] = 6 \cdot 64 = 384$, $m_sc[1] = 5 \cdot 64 = 320$, $m_sc[2] = m_sc[3] = (49 - 11) \cdot (64/2) = 1216$.

The most important steps of the function *level::multigrid_smoothing* are as follows:

- Call *level::build_fsc* to create a subvector of the right hand side vector *fVec*.
- Call *level::apply_Asc_ortho* to apply $A_{sc}^{\perp}$ to u.
- Call *level::solve_gaussian_elimination* to solve the linear system for u_{sc} using the LU-factorization of A_{sc} from the setup.
- Insert the subvector u_{sc} into the overall solution u.

Firstly, the subvector f_{sc} is build from the total right hand side vector *fVec* by iterating over the latter one, and copying only the values corresponding to the given smoother and colour. Therefore, for every entry, the global coordinates within the grid in the r-direction (j) and θ-direction (i) have to be computed as

$$\begin{aligned} j &= ind/ntheta_int, \\ i &= ind - r_index \cdot ntheta_int, \end{aligned} \tag{65}$$

where *ind* is the index of the current entry within the vector *fVec*. For the identification of the smoother and colour of a point the coordinates are then analyzed as presented in Table 7. Using the subvector f_{sc}, we can compute the right side of the equation as

$$f_{total} = f_{sc} - A_{sc}^{\perp} u^{\perp}, \tag{66}$$

using the function *level::apply_Asc_ortho* by applying $\widetilde{A_{sc}^{\perp}}$ on the whole vector u. This function is called for a specific smoother and colour and computes the vector

$$\underset{(m_{sc} \times 1)}{Au} = \underset{(m_{sc} \times m)}{\widetilde{A_{sc}^{\perp}}} \cdot \underset{(m \times 1)}{u} \tag{67}$$

by just just treating the values in u corresponding to the subvector $u^{\perp}$, or equivalently considering $\widetilde{A_{sc}^{\perp}}$ as an extension of $A_{sc}^{\perp}$ where zeros are present in the place of the nonzero entries of A_{sc} from the matrix A.

In principle, the method is similar to the function *level::build_Asc*, starting by an iteration over all points using the coordinates $i \in \{0, \ldots, n_\theta\}$ and $j \in \{0, \ldots, n_r\}$ resp. for θ and r. Differently to *build_Asc*, as *apply_Asc_ortho* is only executed for one individual smoother, we check here if the coordinates of the current point fit together with the given smoother. Analysing the coordinates i and j to be even or odd, see Table 7, the current point is just skipped if it does not belong to the indicated smoother. Subsequently, the local coordinates of the node within the smoother as well as the *index* of the point in the matrix are computed in the same way as in the method *build_Asc* and described in Section 5.2.2. Furthermore, due to $\widetilde{A_{sc}^{\perp}}$ being not square but of size $m_{sc} \times m$, we need two different indices to define the location of a node within the matrix $\widetilde{A_{sc}^{\perp}}$:

- *index*: The index local to the smoother *sc*, computed beforehand (64), indicates the row within the matrix $\widetilde{A_{sc}^{\perp}}$.

- *base_row_index* $= j \cdot ntheta_int + i$: The index global to the whole grid, denotes the index of a node in the whole matrix $A \in \mathbb{R}^{m \times m}$, and serves as column index within the matrix $\widetilde{A_{sc}^{\perp}}$.

Going through the 9 cases of the stencil, the actual column index c of a matrix entry is again defined by the position in the 9-point stencil which denotes the link to a neighbouring point, as defined in Table 10.

Exceptions	PB: add nt to c		PB: subtract nt from c
not for the last row	**top left:** $c = BRI + nt - 1$ for all smoothers,	**top:** $c = BRI + nt$ only for the circle smoothers,	**top right:** $c = BRI + nt + 1$ for all smoothers,
	left: $c = BRI - 1$ only for the radial smoothers,	~~**middle**~~	**right:** $c = BRI + 1$ only for the radial smoothers,
not for the first row	**bottom left:** $c = BRI - nt - 1$ for all smoothers,	**bottom:** $c = BRI - nt$ only for the circle smoothers (And: $r = dc$ for radial smoothers),	**bottom right:** $c = BRI - n + 1$ for all smoothers,

Table 10: Application of the stencil to apply the matrix $A_{sc}^{\perp}$. PB: indicates that a node lies on the periodic boundary. BRI: *base_row_index*. $nt = ntheta_int$, and $dc = delete_circles$.

No top updates are required for the last row ($j = n_r - 1$) and no bottom updates for the first row ($j = 0$ or $j = 1$, resp. when using Dirichlet boundary conditions, and across-the-origin discretztaion at the origin). For any points on the periodic boundary, the number of nodes per line have to be added or subtracted to get the column index and, as there is a link between the last line of the circle smoother and the first line of the radial smoother, there is a bottom update for this line ($j = delete_circles$). Finally, contrary to *build_Asc* where the matrix entries where stored, here, the value is directly summed in the resulting vector Au as follows:

$$Au[index] += val \cdot u[c]. \tag{68}$$

Having computed the right side of the equation f_{total} from (66), the linear system

$$A_{sc}\, u_{sc} = f_{total} \tag{69}$$

can be solved for the vector u_{sc} via forward-backward substitutions. The next step is then to insert this solution vector u_{sc} into the overall solution u, overwriting the current values in u, by iterating over the entries in u_{sc} with the index $k \in [0, m_{sc}[$. In order to know where to insert the elements of u_{sc}, the corresponding index *ind* in the vector u global to the whole grid has to be computed. Before setting this index according to

$$ind = i \cdot ntheta_int + j, \tag{70}$$

the global coordinates i and j of the point in the grid are required. Table 11 shows the computation of these coordinates for the four different smoothers, *n_lines_r* indicating the number of radial lines belonging to the colour of the smoother.

Whenever the implicit extrapolation is activated, the computation of these indices as well as the stencils are modified, which we will explain in detail later.

Smoother	Black $n_lines_r = ceil(nt/2)$	White $n_lines_r = floor(nt/2)$
Circle	$j = 2\,(k/nt)$ $i = k\,\%\,nt$	$j = 2\,(k/nt) + 1$ $i = k\,\%\,nt$
Radial	$j = k/n_lines_r + dc$ $i = 2\,(k\,\%\,n_lines_r)$	$j = k/n_lines_r + dc$ $i = 2\,(k\,\%\,n_lines_r) + 1$

Table 11: Mapping from the local index k of a vector of the smoother sc, to global coordinates (i, j) in the grid. $nt = ntheta_int$, $dc = delete_circles$.

Direct solver

The function *level::facto_gaussian* executes a Gaussian elimination to create a LU-factorization of a matrix A according to algorithm 1. The matrices L and U are both stored in the sparse matrix A, overwriting the latter one with possibly additional entries due to fill in [19]. This function is called for all matrices A_{sc} as well as for the whole matrix A on the coarsest level only once during the setup of the problem in the beginning of the code. To speed up the computations, we hereby skip zero elements during the algorithm, thus using a sparse Gaussian elimination. All the entries in the matrices are ordered in ascending rows, and this information can be used in order to guide the algorithm.

Using this matrix decomposition, the function *level::facto_gaussian* then performs at every iteration of the multigrid cycle forward-backward substitutions conforming to the algorithms 2 and 3 in order to solve a linear system.

Implicit extrapolation

The global parameter *extrapol* indicates whether to use extrapolation, as defined in Section 4.5, within the multigrid cycle or not. Additionally, we have to check for the level, as the extrapolation is only conducted on the finest level ($l == 0$). Our implicit extrapolation consists of two components: the smoothing only on the fine nodes, and the computation of the extrapolated residual.
Concerning the smoothing, we need to know for each node if they are fine or coarse. As a consequence of smoothing only on the fine nodes, the matrix A_{sc} becomes smaller in size due to the extrapolation, and all links of fine to coarse nodes are shifted to the matrix $A_{sc}^{\perp}$. In order to check for a fine or coarse node within the function *level::build_Asc*, we use the variables *level::coarse_nodes_list_r* and *_theta*, which contain a '-1' if all nodes on the corresponding index are fine. Consequently, a point is only treated if it is fine, otherwise it is just skipped in the iteration.

Due to the omitted coarse points, the coordinates of the nodes in the grid (and thus the indices in the matrix) change compared to the smoothing on all nodes. The adapted local coordinates of the nodes (row, col) corresponding to each entry inside the smoother can be obtained from the global coordinates (i, j) in the whole grid by dividing either the row or column coordinate by two, using Table 12. The *index* of a point within the matrix A_{sc}, representing the row index of the entries, is then computed again with (64). Alternatively, we may also count all previously treated fine node as already explained in Section 5.2.2, and use the current number of nodes as *index*. The following computation of the column index c within the matrix stays the same, as well as the update of the points corresponding to the stencil, defined in Table 9. However, for all fine black points, only the middle update is performed while all others are skipped owing to the fact that every such point has two coarse neighbours and these links are shifted to $A_{sc}^{\perp}$.

Smoother	*col*	*row*
Circle	$i/2$	$floor(j/2)$
Radial	$floor(i/2)$	$(j - delete_circles)/2$

Table 12: Mapping from global coordinates (i, j) in the grid, to local coordinates (row, col) for the smoother *sc* when the implicit extrapolation is active on the finest level.

When using the function *level::apply_Asc_ortho*, the matrix is merely applied to the fine nodes of the vector u instead of actually building a subvector and removing the coarse unknowns. Identical to the function *build_Asc*, every node is analyzed whether it is fine or coarse, skipping all coarse nodes. In contrast to the function *build_Asc*, and to the case without extrapolation, the updates corresponding to a link between a fine and a coarse node need to be treated additionally. In fact, for every fine black point, the black circle smoother also executes the left and right updates, whereas the black radial smoother performs the bottom and top updates from Table 10.

In the construction of the subvector f_{sc}, the coarse nodes are actually left out also leading to a vector of smaller size. While the procedure remains unchanged for the two white smoothers, half of the black points need to be skipped in the two black smoothers, due to the coarse points being always black. In the case of the circle black smoother, only uneven coordinates in the θ-direction from $fVec$ are added to $fVec$, and for the radial black smoother, only uneven coordinates in the r-direction are taken.

At the end of the smoothing, the solution vector u_{sc} has to be inserted into the overall solution u again. In order to distribute the entries of u_{sc} over the vector u according to the position of the fine nodes, their global index *ind* within the vector u is computed using (70). The computation of the required global coordinates in the grid i and j is done according to Table 11 using $nt = ntheta_int/2$. Again, only the treatment of the black points has to be adapted in Table 11:

$$\begin{aligned} &\text{black circle smoother:} \quad i = 2\,(k \,\%\, nt) + 1 \\ &\text{black radial smoother:} \quad j = 2\,(k/n_lines_r + delete_circles + ((delete_circles + 1) \,\%\, 2)). \end{aligned} \tag{71}$$

The second decisive part of the implicit extrapolation is the adjustment of the residual computation and restriction on the finest grid. The final *residual* is then computed as

$$\begin{aligned} r_0 &= A_0 \cdot u_0 - f_0, \\ r_1 &= P_{ex}^T r_0, \\ Au_{coarse} &= A_1 \cdot P_{inj}^T \cdot u_0, \\ residual &= 4/3 \cdot res_1 - 1/3 \cdot (f_1 - Au_{coarse}), \\ &= 4/3 \cdot P_{ex}^T \cdot r_0 - 1/3 \cdot (f_1 - A_1 \cdot P_{inj}^T \cdot u_0), \end{aligned} \tag{72}$$

where A_1, f_1 are the operator and the right hand side vector on level $l = 1$. In the first step, the classical residual r_0 of the non-extrapolated system is calculated on the fine level, and restricted to the coarser level as r_1 using *level::apply_prolongation_ex*. Subsequently, a vector Au_{coarse} is computed as intermediate step using the functions *level::apply_prolongation_inj* and *level::apply_A*. Finally, the *residual*, stored in the vector $fVec$ of level 1 in the code, is set to a linear combination of information of the two grids as defined in Section 4.5 in (62). As the initial right hand side vector on level 1 (f_1) is required here, it has to be stored in the beginning of the multigrid cycle to keep it from being overwritten by the restricted residual.

At the end of each multigrid iteration, the residual of the extrapolated system on the finest level can be computed via the function *gmgpolar::compute_residual* using the parameter *extrapol=1*. In comparison to (72), where the residual is used on the coarser grid, in this case, we are interested in the residual on the fine level. Thus, the restriction operator P_{ex}^T is left out and the vector $fVec_1$ as well as the matrix A_1 are prolongated to the fine level using the injection operator P_{inj}. In the end, the residual is computed as:

$$\begin{aligned} r_0 =& f_0^{ex} - A_0^{ex} \cdot u_0, \\ =& [4/3 \cdot f_0 - 1/3 \cdot P_{inj} \cdot fVec_1] - [4/3 \cdot A_0 - 1/3 \cdot (P_{inj} \cdot A_1 \cdot P_{inj}^T)] \cdot u_0, \\ =& 4/3 \cdot res_0 - 1/3 \cdot (\mathbf{P_{inj}} \cdot fVec_1 - \mathbf{P_{inj}} \cdot A_1 \cdot P_{inj}^T \cdot u_0). \end{aligned} \tag{73}$$

Using all of these elements in the C++ code, we obtain a functional implementation. In the next section, we compare the numerical results from the initial implementation, as used in [39], to our new implementation in terms of solution accuracy and execution time.

5.3 Numerical experiments

The initial goal of this thesis was the reproduction of the results from paper [39] which were produced with the help of the formerly existing matlab code. Evaluating several test cases for different geometries, inner radii and discretizations at the origin, in fact, the same results could be generated by the new C++ code.

In this section, the results of the multigrid solver are presented for a specific test problem, as defined in Section 4.1. We base our computations on a specific manufactured solution u, and the corresponding right hand side of the equation for a representation on the Shafranov geometry. This solution has two interesting properties which are challenging for the multigrid solver: it is oscillating and not aligned with the polar grid [36]. The solution is expressed as

$$u(x,y) = (1.3^2 - r^2(x,y))\cos(2\pi x)\sin(2\pi y). \tag{74}$$

Moreover, in this Section , we use the Shafranov geometry and an innermost circle around the origin of a radius $r_0 = 10^{-5}$. The grid is defined using $nr_exp = 4$ and $fac_ani = 3$, resulting in 49×64 nodes, and then refined again three times in order to get consecutive grids with an increasing number of unknowns.

We carry out our experiments on a computer at the FAU Erlangen-Nuremberg with a Linux operating system, a memory size (RAM) of $64\,GiB$, one socket with 4 cores, and 8 threads in total. The timings may be not quite representative as they are not run on a cluster, and hence, background processes could not be definitely avoided.

The mean residual reduction factor is defined as

$$\hat{\rho} = \sqrt[its]{\frac{||r_0^{its}||_2}{||r_0^0||_2}}, \tag{75}$$

using the residuals on the finest level ($l = 0$) in the 2-norm, from the last compared to the first iteration [39]. Solving the same problem on four consecutive refined grids with different size m, the error reduction order is computed by a comparison of the corresponding error norms. Using the beforehand computed weighted 2-norm or ∞-norm of the final error vectors of the current (k) and the previous ($k-1$) grid

$$\frac{||e_{k-1}||}{||e_k||} = \left(\sqrt{\frac{m_k}{m_{k-1}}}\right)^{ord}, \tag{76}$$

we can solve the equation for the order

$$ord = \frac{log(\frac{||e_{k-1}||}{||e_k||})}{log(\sqrt{\frac{m_k}{m_{k-1}}})}, \tag{77}$$

see [39].

In Table 13, we present the results from running the C++ implementation of GmgPolar on all problem sizes with and without implicit extrapolation activated. For every run of the solver, the number of iterations (its), the mean residual reduction factor $\hat{\rho}$, the error in the scaled 2-norm and in the ∞-norm together with the corresponding reduction order as well as the run time of the solver are displayed.

Since using a Dirichlet boundary at the origin requires the knowledge of the exact solution, this treatment of the singularity cannot be used for a real world application problem. Thus, the approach of discretizing across the origin has been established as alternative and investigated regarding its suitability. The obtained results are identical to the configuration in [39] with Dirichlet boundary conditions if $r_0 \to 0$. For r_0 being too large, for instance $r_0 = 10^{-2}$, we get unsatisfactory results. Moreover, obtaining the most considerable deviations from the exact solution around the origin, we can observe that the error in the ∞-norm may take higher values than in the 2-norm, being more extensively influenced of a large error at a single point.

Considering the error reduction order of the multigrid solver without extrapolation, we achieve up to quadratic convergence in both norms as expected. In the case of integrating implicit extrapolation into the algorithm, fourth order would be achieved for simpler geometries without any

$n_r \times n_\theta$	its	$\widehat{\rho}$	$\|\|err\|\|_2/sqrt(m)$	ord	$\|err\|_\infty$	ord	$time[s]$
Multigrid without extrapolation							
49×64	46	0.67	$7.1 \cdot 10^{-2}$	-	$1.4 \cdot 10^{-1}$	-	0.12
97×128	45	0.66	$1.8 \cdot 10^{-2}$	2.0	$4.1 \cdot 10^{-2}$	1.9	0.87
193×256	44	0.66	$4.6 \cdot 10^{-3}$	2.0	$1.1 \cdot 10^{-2}$	2.0	10.73
385×512	44	0.65	$1.1 \cdot 10^{-3}$	2.0	$2.6 \cdot 10^{-3}$	2.0	164.02
Multigrid with implicit extrapolation							
$n_r \times n_\theta$	its	$\widehat{\rho}$	$\|\|err\|\|_2/sqrt(m)$	ord	$\|err\|_\infty$	ord	$time[s]$
49×64	73	0.77	$7.5 \cdot 10^{-3}$	-	$2.6 \cdot 10^{-2}$	-	0.16
97×128	77	0.79	$5.6 \cdot 10^{-4}$	3.8	$3.0 \cdot 10^{-3}$	3.2	0.79
193×256	78	0.79	$4.2 \cdot 10^{-5}$	3.7	$3.6 \cdot 10^{-4}$	3.0	7.19
385×512	78	0.79	$3.0 \cdot 10^{-6}$	3.7	$4.5 \cdot 10^{-5}$	3.0	98.81

Table 13: Results of the multigrid solver, for a test problem with Dirichlet boundary conditions on the inner radius (10^{-5}) and outer radius for the Shafranov geometry. Iteration count its, mean residual reduction factor $\widehat{\rho}$, errors of the iterative to exact solution evaluated at the nodes in the weighted 2- and ∞-norms with corresponding error reduction order.

singularities and additional refinement. In our case, the symmetry of the grid is not completely maintained and thus, some 3rd order error terms may not be canceled out entirely. Consequently, we get a cubic convergence order in the ∞-norm and an order between 3.5 and 4.0 in the 2-norm [39].

6 Improving the solver GmgPolar

We have introduced the different aspects of the solver *GmgPolar* in Section 4 as well as the details of the C++ implementation in Section 5. The preliminary performance of the solver was shown in the previous Section 5.3 where the same accuracy but a faster execution are obtained compared to the Matlab implementation. In order to improve the efficiency of *GmgPolar*, different possibilities exist.

In this Section, some optimizations that were applied in the course of the thesis are presented. First, we improve the implementation, in particular the application of the matrices (operator, prolongation, smoother), in order to make the best use of modern computing architectures (vectorization, etc.). Secondly, it is proposed to modify the smoother in order to improve the potential for parallelism. Thirdly, an alternative extrapolation scheme is studied which combines fast convergence (similar to no extrapolation) and a high order approximation (similar to the extrapolation). Then, we parallelise the GmgPolar solver using an OpenMP multitreaded parallelism, and GPU parallelism. Finally, some future possibilities to optimise the solver further are suggested.

6.1 Code optimization

The preliminary implementation of the multigrid solver was a literal translation of the Matlab code, and the algorithms used could be said to be naive. Here, our goal is to improve the computational efficiency of the solver through an optimization of its different components with respect to the computing architecture.

The code optimizations mainly focus on the construction and application of the different operators, i.e the functions *build_A*, *apply_A*, *build_rhs*, *build_Asc*, *apply_Asc_ortho*, *apply_prolongation* from class *level*. The implementation of these optimised functions were conducted by my mentor Dr. Leleux. In terms of optimization, there are two aspects we consider. First, the number of operations within the functions must be decreased. Secondly, on modern computing architectures, it is possible to perform more than one instruction during each clock cycle of a core in the computing processor. We say that we increase the number of IPC (*instruction per cycle*). This effect is made possible by the combination of several modern technologies, in particular vectorized processors.

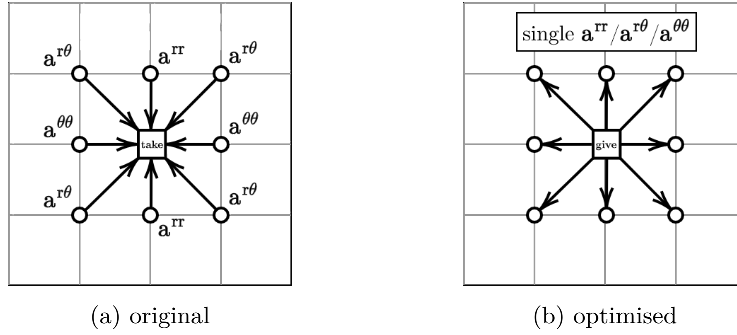
Let's consider the example of the application of the operator A (*level::apply_A*). The principal difference between the two versions is then the following:

- In the original implementation, when iterating over the grid, the stencil is applied to one point after another computing the values of α, $det(DF)^{-1}$, a_{rr}, a_{rt} and a_{tt} of the surrounding points, see Figure 26a. Consequently, those coefficients have to be computed several times throughout the iteration, serving as different links (top, bottom, right, ...) to distinct nodes.
- In the optimised implementation, all coefficients are computed for the current point in the iteration and then given to the neighbouring nodes for their update, see Figure 26b. In other words, the computation of those values is done only once for every point and thus, the stencil updates are not conducted in one integrated step but split up nine separated sub-steps.

By avoiding redundant coefficient computations, we decrease the overall number of operations required. Additionally, in the optimised implementation, we can compute easily the coefficients for a whole line of points at the same time, thus vectorizing the corresponding operations and improving the computational efficiency even further.

In order to demonstrate the positive effect of this optimization on the method *level::apply_A*, we apply it on 10 random vectors and measure the execution time. Figure 27 shows the resulting execution time for vectors of size increasing from $m = 40$ to $m = 9.4 \cdot 10^6$. We observe that the execution time of the optimised implementation is 100 times lower in average. Also, the execution time increases linearly with the problem size, which empirically confirms the linear complexity of applying this operator, which was introduced in [22].

The principle of the optimization for *apply_A* is directly applied to optimise the functions *build_A*, *build_rhs*, *build_Asc*, and *apply_Asc_ortho*. Concerning the function *apply_prolongation*, we use our knowledge on the grid coarsening and on the stencil of the prolongation operator, see



(a) original (b) optimised

Figure 26: Principle behind the application of the operator A. In the original version, for each point we compute the coefficient of neighbouring points to update the local operator entries. In the optimised version, for each point, the local coefficients are computed to update the operator entries of neighbouring points.

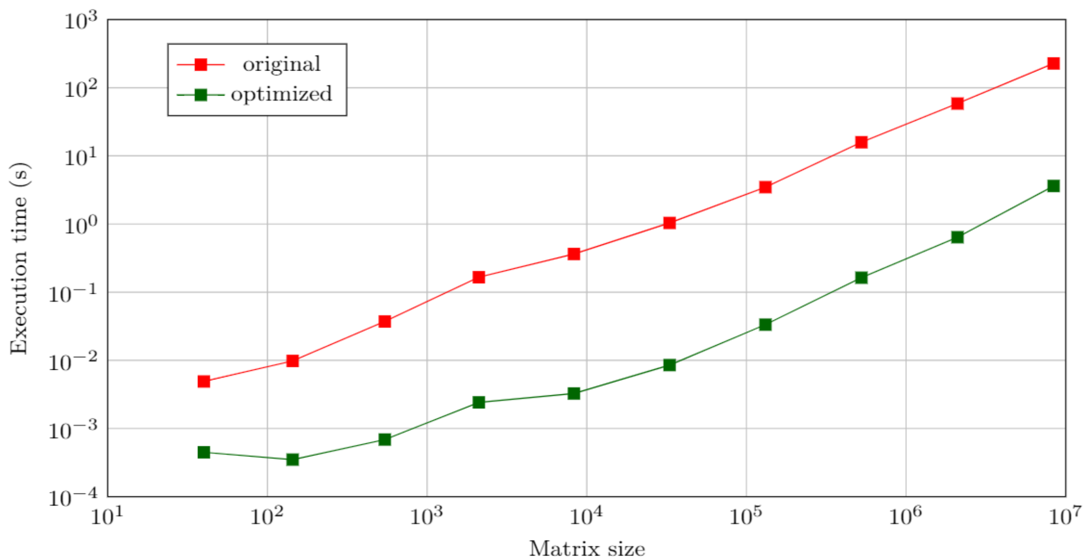


Figure 27: Comparison of the run time of the method *level::apply_A* of the original and the optimised version.

Section 4.3, in order to vectorize its application in a simple way. The latter optimization is very straight-forward and we do not detail it here.

Another performance optimization has been applied to the LU-factorization and the direct solver, which were a significant bottleneck of the whole algorithm. The first implementation was a direct writing of the Gaussian elimination algorithm, by accessing each entry $A(i, j)$ of the matrix with functions *level::get_element* and *level::set_element* which search through all non-zero entries of A. These two functions were running really slow due to the repeated iteration over the whole matrix. Both were completely left out in the optimised versions because an iteration over the whole matrix is not necessary as the matrix entries are sorted by ascending rows and, as the number of elements per row is exactly known, they can be directly accessed. Alternatively, we included the possibility to use the external sparse direct solver *MUMPS* [46, 2] in *GmgPolar*.

In the beginning of the coding process, when only focusing on computing the correct approximations, we still had run times for the whole solver about 1 000 to 2 000 seconds for a test problem of size 49×64. In the course of the optimizations, the run time was then reduced down to only an eighth of one second. In Table 14, the reduction of the run time at specific dates throughout the optimization process can be seen for a test problem and different grid sizes. The specificities of

each version are:

- 11.08.2021: The first working code version is a direct C++ translation of the Matlab code. There is still no option for implicit extrapolation, and only the direct solver had been optimised, yet.
- 01.09.2021: The second version, including the extrapolation, is already optimised with respect to the application of the matrices A and P.
- 11.10.2021: Finally, this version contains all major performance optimizations.

$\mathbf{n_r} \times \mathbf{n_\theta}$	**Multigrid without extrapolation**			**Multigrid with extrapolation**	
	11.08.21	01.09.21	11.10.21	01.09.21	11.10.21
49×64	1.45	0.53	0.12	0.89	0.16
97×128	6.00	2.59	0.87	5.83	0.79
193×256	31.66	18.03	10.73	51.19	7.26
385×512	268.75	198.95	164.02	688.63	101.93

Table 14: Run time (in s) of the code versions for test problems of increasing size using the Shafranov geometry, with Dirichlet boundary conditions on the inner radius ($r_0 = 10^{-5}$) and outer radius.

6.2 Decoupled circle-radial smoothing

In Section 4.4.2, we introduced the combined circle-radial smoother used in *GmgPolar*, which we call *coupled circle-radial smoothing.* At each iteration of this smoother, detailed in the paper [39], we successively apply the circle black, circle white, radial black and finally radial white block smoothers. In the course of coding, it occurred that this smoothing procedure was not ideal in terms of parallelism potential. In this section, we introduce an alternative smoothing procedure called *decoupled circle-radial smoothing.* In the code, the smoothers *coupled* and *decoupled circle-radial* are activated resp. with the parameter $smoother = 3$ and $smoother = 13$.

For both smoothing procedures, we follow [39] and first smooth on the black unknowns before treating the white points. When relaxing on the white unknowns, the updated approximations of the neighbouring lines are directly used [64, chapt. 5], i.e., the results from the corresponding black smoother are included into the vector u_{sc}, which is used in the application of the matrix $A_{sc}^{\perp}$ for the white nodes. Consequently, the relaxation procedure on the black and white nodes is done, sequentially because the white smoothing step depends on the previously executed smoothing of the black points [39].

The dissimilarity of the two versions, in fact, lies in the interaction of the two circle smoothers with respect to the two radial smoothers. The interior boundary, representing the switch from the circle to the radial smoother, actually only affects the points next to it, thus all nodes in the last circle line as well as ones positioned at the first r-coordinate of the radial smoother.

As stated above and described in [39], originally the circle and radial smoothers are applied successively, resulting in a block Gauss-Seidel like iteration. Thus, in order to update the points next to the interior boundary, the values of the previous iteration are directly used for the other smoother. We alternate between circle and radial smoothers, thus the name *coupled circle-radial smoothing.* The challenge hereby lies in a later scheduled parallel version, for which we propose to decouple the circle and radial smoothers. In this new relaxation procedure, at each iteration of the smoothing, the circle and radial smoothers are simultaneously applied on the same approximated solution, thus resulting in a block Jacobi like iteration. Potential for a better parallelism then comes from the fact that both smoothers can be applied independently in parallel.

In order to implement the *coupled circle-radial smoothing* in C++, the four smoothers are just called one after the other and the solution vector u_{sc} is inserted into the solution u right after every smoother. Owing to the fact that all four smoothers directly insert their sub-solution u_{sc} into the vector u, the updated unknowns are already used in the following smoothers. However,

the parallelisation of this version is quite challenging as the radial smoothers use information from the circle smoothers [39]

Actually envisaged as parallel procedure, the sequential implementation of the *decoupled circle-radial smoothing* whereas is more complex. In a consecutive application, it has to be ensured, that the radial smoothers do not already use the results from the circle smoothers. Therefore, the solution from the previous iteration needs to be stored for use by each smoother. Consequently, two vectors *u_previous_c* and *u_previous_r*, corresponding resp. to the circle and the radial smoothers, are both set to the current solution vector u in the beginning of the whole smoothing procedure. Subsequently, the two smoothers operate on their own version of the vector enforcing also the method *apply_Asc_ortho* to be applied on *u_previous* instead of on u. After the smoothing procedure, the solution vector u is created from the two emerged vectors, using *u_previous_c* for all circle points and *u_previous_r* respectively for the radial unknowns. This version now has a good potential for parallelisation. While the white smoothers depend on the the already updated values from the corresponding black smoothers, the two circle smoothers could work in parallel with the two radial smoothers, using the results from the previous iteration for the interior boundary. Nevertheless, the processes have to wait for at least the previous iteration of the other smoother to be finished in order to use these values for their updates.

Table 15 shows the number of iterations and execution time obtained from applying both smoothing procedures sequentially on problems of increasing sizes, when using the Shafranov geometry with $r_0 = 10^{-5}$ and Dirichlet boundary conditions on the inner and outer boundaries. We observe that the number of iterations doubles when using the *decoupled circle-radial smoothing*, compared to the *coupled* version. This is expected since there is less exchange of information between the two kind of smoothers when using the *decoupled* version. While the *decoupled* version can be more easily parallelised, the *coupled* version convergences in half of the number of iterations.

	Coupled circle-radial smoothing		**Decoupled circle-radial smoothing**	
$n_r \times n_\theta$	its	time (s)	its	time (s)
49×64	46	0.12	80	0.18
97×128	45	0.87	75	1.06
193×256	44	10.73	72	11.47
385×512	44	164.02	70	168.92

Table 15: Iteration count and run times when applying GmgPolar with the two different smoothing procedures, for a test problem of increasing sizes using the Shafranov geometry, with Dirichlet boundary conditions on the inner radius ($r_0 = 10^{-5}$) and outer radius.

However, converging in considerably less iterations, we could investigate a workaround in order to parallelise the *combined circle-radial smoothing*. As already mentioned, the link between the two circle and the two radial smoothers only affects the points right next to the interior boundary. Moreover, executing the circle smoothers first, only the updates of the radial points depend on the previously conducted smoothing of the last circle line. When starting the procedure with smoothing the colour corresponding to the last circle line, the new information could be directly used by the radial smoothers. Consequently, the radial smoothers then only have to wait until half of the circle smoothing procedure is done and can be conducted in parallel with the second half of the circle lines. The smoothing of the radial lines, being approximately of half the length than the circle lines, then is assumed to finish in almost the same time than the smoothing of the remaining circle lines.

Due to time constraints, results from the actual parallelisation of both smoothing procedures could not be obtained and should be the subject of future research, in order to find out in which cases each of the smoother should be used.

Finally, one could try to parallelise the smoothing even further by decoupling the black and white smoothers for the radial and circle smoothers. However, we expect a drastic degradation of the convergence since all white points would lose the updated information of the neighbouring black points. In the *decoupled circle-radial smoothing*, only the update of the points close to the switch between circle and radial smoothers was affected, thus a relatively small degradation of the convergence. Therefore, we do not investigate this any further here.

6.3 Implicit extrapolation with full grid smoothing

This Section is mainly based on a private communication with Prof. Rüde [57] as well as the extensive research work on extrapolation techniques for multigrid methods contained in [9, 28, 56, 34].

The application of implicit extrapolation to the multigrid algorithm results in an improved order of accuracy, see Section 4.5. To be more precise, while our discretization scheme allows to get an error in the order of h^2, when using implicit extrapolation, we may achieve an error in the order of up to h^4. On the other hand, due to the smoothing on the fine nodes only, information requires more time to travel through the grid, than in the case of smoothing on all grid nodes. Therefore, the algebraic convergence of multigrid with implicit extrapolation is slower than of standard multigrid.

The idea in this context, in order to improve the convergence, is the combination of standard multigrid with extrapolated multigrid. Inspired from the discussion at the end of Section 4 in [56], we propose here to use our classical smoother applied on the full grid from the standard multigrid, as introduced in Section 4.4.2, while computing and restricting the residual according to the theory of the prescribed implicit extrapolation [56]. Here, we call this approach *implicit extrapolation with full grid smoothing*.

As stated in [45]:

> "This leads to an algorithm that is paradoxically based on two competitive iterations with different fixed points".

In other words, the full grid smoothing resolves the high frequencies of the solution and converges to a low order h^2 solution. The extrapolated coarse-grid-correction whereas contributes the low frequencies, and converges to the more accurate, high order h^4 solution. The described mixture of these two iterations then will obviously converge to something in between the h^2 and the h^4 solution. However, as detailed in [28], it turns out that the sequence rather converges to the h^4 solution. Consequently, the resulting solution has a slightly higher error compared to the classical implicit extrapolation but, on the other hand, the performance of the solver is expected to improve significantly and to converge much faster, thanks to the smoother also working on coarse nodes. The loss in accuracy should not be much worse than the discretization error itself [56].

Owing to the fact that the solution now consists of two components with different quality, we do not have one fixed matrix system anymore, but two different ones. Since the solution is positioned somewhere in between, neither of the residuals of both systems will be close to zero and, thus, the convergence of the method cannot be measured by the residual anymore. Using the backward error

$$backward\ error = \frac{||b - Ax||_\infty}{||A||_\infty \cdot ||x||_1 + ||b||_\infty} \tag{78}$$

also does not lead to satisfactory results. Instead, the scaled 2-norm of the error vector $||e||_2/\sqrt{m}$ is used as convergence criterion, m being the size of the matrix. However, we cannot define a fixed threshold on this norm for using it as stopping criterion. Indeed, the error depends on the order of the solution that we reach in the end, which again depends i.e. on the geometry, the grid size, the boundary conditions and on the parameters of the multigrid cycle (number of smoothing steps, cycle type, etc.). Consequently, the iteration should stop as soon as the error norm does not decrease any further and, thus, the last two consecutive iterates do not differ anymore. We can either use the relative error norm to the last iteration

$$\frac{||e_k||_2}{||e_{k-1}||_2} >= 1 \tag{79}$$

as convergence criterion, or the difference between the norms, scaled by the initial error

$$\frac{abs\left(||e_k||_2 - ||e_{k+1}||_2\right)}{||e_0||_2} < 10^{-8}. \tag{80}$$

In both cases, the scaling by $\sqrt{m}$ cancels out. In our case, knowing the exact solution of the PDE problem, we choose the criterion based on difference of the error norms (80). However, in a real world application problem, the exact solution is not known a priori and thus, the error cannot be used as convergence criterion in such a case.

In the code, this alternative variant of the implicit extrapolation can be activated using the parameter *extrapol=2*. Hence, we merely adapt the if-conditions of the implicit extrapolation so that the adjusted smoothing procedure is only executed in the case of $extrapol == 1$ and the extrapolated coarse-grid-correction for $extrapol \geq 0$.

In Table 16, the performance of the alternative extrapolation version with full grid smoothing is compared to the multigrid solvers with standard implicit extrapolation and without extrapolation. It can be observed that, as expected, the order of the error of the new variant is in between the ones of the two other solvers, but closer to the higher order obtained with implicit extrapolation. Furthermore, less iterations are required than in the case of implicit extrapolation, meaning that the scheme tends to converge faster to the higher order solution. Surprisingly, we observe that the number of iterations is even lower than in the case of multigrid without extrapolation. A conceivable reason is that the comparison of the iteration count is actually just not reasonably, using another convergence criterion for the implicit extrapolation with full grid smoothing. A future investigation for further reasons as well as the analysis of alternative convergence criteria which do not require the exact solution could be the work of an entire PhD.

nr × ntheta	its	$\|\|e\|\|_2/sqrt(m)$	ord	$\|\|e\|\|_\infty$	ord	time(s)
		No extrapolation				
49 × 64	46	$7.1 \cdot 10^{-2}$	–	$1.5 \cdot 10^{-1}$	–	0.12
97 × 128	45	$1.8 \cdot 10^{-2}$	1.99	$4.1 \cdot 10^{-2}$	1.89	0.87
193 × 256	44	$4.6 \cdot 10^{-3}$	1.98	$1.1 \cdot 10^{-2}$	1.91	10.73
385 × 512	44	$1.1 \cdot 10^{-3}$	2.07	$2.6 \cdot 10^{-3}$	2.08	164.02
		Implicit extrapolation				
49 × 64	73	$7.6 \cdot 10^{-3}$	–	$2.6 \cdot 10^{-2}$	–	0.16
97 × 128	77	$5.6 \cdot 10^{-4}$	3.79	$2.9 \cdot 10^{-3}$	3.19	0.79
193 × 256	78	$4.2 \cdot 10^{-5}$	3.75	$3.6 \cdot 10^{-4}$	3.02	7.26
385 × 512	78	$3.0 \cdot 10^{-6}$	3.00	$4.5 \cdot 10^{-5}$	3.01	101.93
		Implicit extrapolation and full grid smoothing				
49 × 64	35	$1.3 \cdot 10^{-2}$	–	$5.5 \cdot 10^{-2}$	–	0.12
97 × 128	33	$1.4 \cdot 10^{-3}$	3.24	$8.3 \cdot 10^{-3}$	2.75	0.87
193 × 256	32	$1.2 \cdot 10^{-4}$	3.56	$8.7 \cdot 10^{-4}$	3.27	11.03
385 × 512	31	$9.0 \cdot 10^{-6}$	3.74	$7.1 \cdot 10^{-5}$	3.62	169.72

Table 16: Iteration count and error norms of the different extrapolation variants, for a test problem with Dirichlet boundary conditions on the inner radius (10^{-5}) and outer radius for the Shafranov geometry, using the circle-radial coupled smoothing version. Notations as in Table 13.

6.4 Parallelisation

As the size of problems resulting from the simulation of real-life applications increases, such as the ones constructed in the context of plasma simulation, parallelisation becomes necessary and is enabled by modern supercomputer centers on the road to exascale computing. The primary goal of parallel computing is the acceleration of computations which is achieved by dividing large problems into smaller ones and solving them simultaneously [14, chapt. 1].

As already mentioned for *GmgPolar*, the circle and radial smoothers might be applied in parallel, depending on the variant of the smoother. Additionally, since using a 9-point stencil of length one, black lines are pair-wise independent from each other and so are white lines for both smoothers [22]. Following, all lines with the same colour can be treated simultaneously, resulting in a very high potential of the code for parallelism [22].

Furthermore, the workload of single methods applying for instance a matrix to a vector can be distributed onto several processes, too. In the course of this thesis, the function *level::apply_A*, which applies the whole matrix A to a vector, was chosen for parallelisation. In order to be integrated in the simulation code GyselaX in the future, we propose here to use OpenMP parallelisation, see Section 6.4.1. It has to be mentioned, that the parallelisation of this function was only intended

as a proof-of-concept of the potential for parallelisation, but the parallelisation of the whole solver is still out of the scope of this thesis. Moreover, a simplified code version of the GmgPolar solver without extrapolation has been ported to GPUs and parallelised with Cuda by the team throughout the *Hackathon CSCS 2021*, see Section 6.4.2.

6.4.1 Parallelisation with OpenMP

OpenMP is a parallel programming model for shared memory parallelism [13] consisting of a set of compiler directives, and a library of support functions [51, chapt. 17]. In this Section we detail the principle behind our parallelisation of the function *apply_A* using OpenMP, and task-based scheduling.

The function *level::apply_A* mainly consists of two nested for-loops (for the r- and θ-directions) to iterate over all grid points, in order to apply the matrix A to a vector u, by adding the computed values to the vector entries of Au. The simplest approach for the parallelisation of such a repeated application of a binary operator is the use of *#pragma omp parallel for reduction* [13, chapt. 3] in order to:

1. *parallel for:* distribute the iterations of the outer loop, corresponding to each radii, to the OpenMP threads,

2. *reduction:* sum the resulting local values of the vector Au.

However, the reduction clause can only be applied to single variables or arrays but not to *std::vectors*, which we use in the code for the representation of the vector Au.

Another option is the usage of so called *tasks* - sequential code fragments which are executable by one thread [29, chapt. 6]. Hereby, the workload is divided into several tasks which are then distributed at run-time to the different threads to be processed concurrently [13, chapt. 4]. Using the update of each radius of the grid as a separate task group, we have to be careful that the distinct groups do not disturb each other. Since there is no implicit synchronisation established between the distinct tasks, the programmer has to ensure by himself that no dependencies exist between the task groups [29, chapt. 6]. Owing to the beforehand applied performance optimization, when iterating over the grid, all neighbouring entries of a current point may be changed. Thus, when treating one specific point (r_i, θ_j) in the matrix, also the values of the neighbouring points $(r_{i\pm1}, \theta_{j\pm1})$ are updated. If one value is accidentally accessed to be modified by more than one thread at the same time, we get a so called *race condition*. This phenomena occurs in case of multiple threads accessing a shared variable at the same time, with at least one of them by writing [51, chapt. 17]. As the result of the computation truly depends on the order of the executions [29, chapt. 3] while having no guaranteed ordering among the multiple threads [13, chapt. 2], the computation may exhibit a non-deterministic behaviour [51, chapt. 17]. Using the optimised application of A introduced in Section 6.1, Figure 28 shows the conflicts resulting from the simultaneous update of two consecutive radii in the absence of synchronization mechanisms.

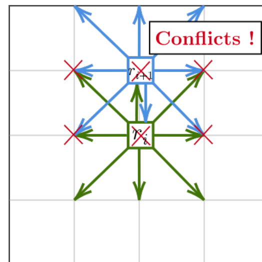


Figure 28: Conflicts in red cross resulting from the parallel update of two consecutive radii r_i and r_{i+1} in the function *apply_A*.

Consequently, such access on a shared variable in a parallel region must be controlled through some form of synchronization that allows a thread exclusive access to it [13, chapt. 2]. Therefore, we can only treat every third line in parallel because handling one single point may update values from

three different lines in total. Having completed the treatment of every third line, we can start with the next bunch of lines, splitting the whole application of the matrix into three parts, according to Figure 29. Hereby, we use the statement *#pragma omp taskwait* after each of the three parts to wait for the previous task group to be finished before starting with the next one.

Task 1	0	3	6	9	···
Task 2	1	4	7	10	···
Task 3	2	5	8	12	···

Figure 29: Splitting of the workload in three different parts, where each task group is independent.

The next step is to get rid of the *#pragma omp taskwait* which enforce a synchronization of all threads and can lower the parallel performance. We thus need to implement a mechanism for the task synchronisation [29, chapt. 6] by inserting task dependencies (available since OpenMP-4.0) which enforce additional constraints on the scheduling of tasks or loop iterations by the creation of a *directed acyclic graph* (DAG) of dependencies [50]. Therefore, the treatment of one specific radial line in the grid can start as soon as all dependencies have been finished. As shown in Figure 30, we start with the first task group executing all radial lines with index 0, 3, 6, 9, and so on. Then, the second task group does not need to wait until the whole first task group is done, but only for the underlying blocks of grid lines. For instance, line 1 can be treated as soon as line 0 and 3 are done and, respectively, line 2 only needs to wait for the lines 1 and 4. In fact, considering the integers $0 \leq i < n_r/3$, we define three rules for the dependencies:

- radius $3i$ is independent of all other radii,
- radius $3i + 1$ depends radii $3i$ and $3i + 2$,
- radius $3i + 2$ depends radii $3i + 1$ and $3i + 3$.

Task 3	2	5	8	···		
Task 2	1	4	7	10	···	
Task 1	0	3	6	9	12	···

Figure 30: Dependencies of the OpenMP tasks. The relations are read from bottom to top.

The following code fragment presents the structure of the parallelised function *apply_A* using tasks and dependencies as described above. In C++, task dependencies can only be defined based on the address of variables, and the array *dep* hereby only contains a list of addresses which are used to control the access on the distinct matrix rows.

```
void level::apply_A(std::vector<double> u, std::vector<double>& Au) {
    int* dep = new int[nr];
    int start_j = 0;
    #pragma omp parallel shared(dep){
            #pragma omp single {
                #pragma omp task {
                    //possibly treat the inner Dirichlet boundary
                    start_j = 1;
                }
                #pragma omp task depend(out: dep[start_j]) {
                    //treat the first line
                }
                for(int j=start_j + 3; j < nr_int; j += 3) {
                    #pragma omp task firstprivate(j) depend(out: dep[j]) {
                        for(int i = 0; i < ntheta_int; i++) {
                            //treat the interior part 1
                        }
                    }
                }
                For{int j=start_j+1; j<nr_int; j+=3 }{
                    #pragma omp task firstprivate(j) depend(in: dep[j-1])
                                    depend(out: dep[j]) {
                        For{int i=0; i<ntheta\_int; i++}{
                            //treat the interior part 2
                        }
                    }
                }
                For{int j=start_j+2; j<nr_int; j+=3 }{
                    #pragma omp task firstprivate(j)
                                    depend(in: dep[j-1], dep[j+1])
                                    depend(out: dep[j]){
                        For{int i=0; i<ntheta_int; i++}{
                            //treat the interior part 1
                        }
                    }
                }
                #pragma omp task depend(in: dep[nr_int-2])
                                    depend(out: dep[nr_int-1]){
                    //treat the last line
                }
                #pragma omp task depend(out: dep[nr_int]){
                    //treat the outer Dirichlet boundary
                }
            }
        }
}
```

Hereby, we use one *#pragma omp parallel* statement for the definition of a general parallel region around the whole function to express parallel execution [13, chapt. 4]. Moreover, *#pragma omp single* is used once in the beginning, having only one single thread entering this region, reading the code and creating all tasks as they are defined. The other threads then repeatedly fetch and execute the tasks as they appear in the shared task queue [13, chapt. 4]. When defining the tasks, depend-clauses *depend(dependence-type: list)* are used which are specified by the dependence-type (*in/out*) and a list of dependent storage locations. If the storage location of one item within the list of an *in*-type clause is the same as of a list item from another depend clause with an *out*-type, then this task will be dependent on the previously generated sibling task [50]. Using the *firstprivate*-clause in addition, the compiler allocates a private copy of the specified variable for each thread executing the block [51, chapt. 17]. Furthermore, all private instances are initialized to the value of the masters copy [13, chapt. 3], inheriting the value of the shared variable instead of being undefined [51, chapt. 17]. Additionally, all globally used variables such as coefficients or indices need to be redefined as local instances inside every task so that they can be accessed and modified at the same time within distinct tasks. We define the speed-up of a program as

$$\textit{speed-up} = \frac{t_{sequential}}{t_{parallel}}. \tag{81}$$

We now run the final parallel version of the function *apply_A* on 10 random vectors of size $m = 2.1 \cdot 10^6$ and $m = 8.4 \cdot 10^6$, and show the resulting speed-up for different numbers of threads in

Figure 31. Since the run-time of codes, in particular parallel codes, can vary from one execution to another, several runs of the program are required using lastly the mean value of the run times as representative time.

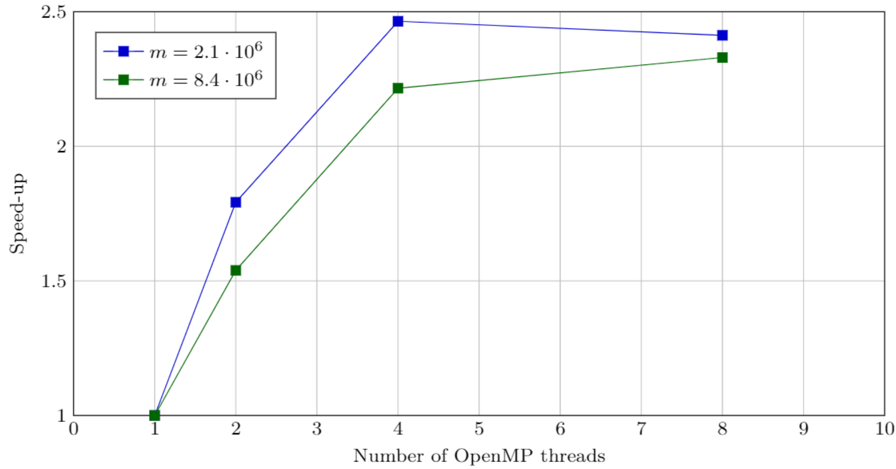


Figure 31: Speed-up of the function *level::apply_A* using OpenMP parallelisation for two different grid sizes m.

Based on the parallelisation of the function *build_A*, the functions *build_rhs*, *build_Asc*, and *apply_Asc_ortho* can be adapted, too. Concerning the function *apply_prolongation*, when applying P to a coarse vector u_c, all resulting entries of Pu_c can be treated independently and thus, the parallelisation is much simpler, not requiring any task dependence mechanism.

Eventually, we have to use some monitoring tool in order to analyze the workload of the different processes over time and to get a satisfactory scaling of the algorithm. This, however, goes beyond the course of this thesis.

6.4.2 Parallelisation with GPUs and Cuda

Over the last decades, high-performance computing has evolved significantly, particularly because of the emergence of heterogeneous architectures [14, chapt. 1], containing both central (CPUs) and graphics processing units (GPUs) [58, chapt. 1]. GPUs, being multithreaded graphical interfaces [58, chapt. 1] which are optimised to perform hundreds of computations in parallel [58, chapt. 4], have been originally designed to perform specialized graphics computations. However, they have recently become more powerful and generalised, enabling them to be applied to arbitrary "general-purpose parallel computing tasks with excellent performance and high power efficiency" [14, chapt. 1]. On a heterogeneous platform, an application is typically initialized by the CPU, which is responsible for control intensive, serial tasks such as the management of the environment, code and data. With the GPUs executing the compute-intensive, data-parallel workload, large data sets are distributed across multiple cores and operated on at the same time. This conjunction is a very powerful combination [14, chapt. 1].

The CUDA architechture by NVIDIA is a general-purpose parallel computing platform and programming model [14, chapt. 1] which aims at making GPUs more convenient to use for general-purpose computations [58, chapt. 1]. With just some small extensions to the standard C programming language [14, chapt. 2], the CUDA C language helps to improve the performance and programmer productivity on heterogenous architechtures enabling the simple access of the GPU for arbitrary computations [14, chapt. 1].

Within this context, we call the CPU with the system's memory the *host* and the GPU with its attached memory the *device*. A function defined in a *.cu*-file and intended to be executed on the device but called from the host is called a *kernel* and qualified with the keyword *__global__* [58, chapt. 3]. The *__device__*-qualifier on the other hand indicates a function which is called from and executed on the device [14, chapt. 2]. When using the CUDA runtime, kernels can be invoked

in-line on the device with a special triple-angle-bracket syntax [67, chapt. 3] and executed by an array of threads all running the same code [14, chapt. 1]. A kernel is called from a host thread by the syntax *kernel_name <<<gridsize, blocksize >>> (parameters)* [67, chapt. 7] using a numerical tuple of parameters that will influence how the runtime will launch the device code [58, chapt. 3]. The grid, which is used to execute our kernel, is specified by the *gridsize* and consists of several parallel blocks of dimension *blocksize* containing multiple threads [67, chapt. 7]. In order to use memory on the GPU it has to be allocated via *cudaMalloc()* on the device, taking as arguments a pointer to the pointer holding the address of the memory as well as the size of the allocation. After the computations, obviously we need to free this allocated memory again using *cudaFree()* [58, chapt. 3]. As host pointers can only access memory from the host code and device pointers respectively from device code [58, chapt. 3], data must be copied explicitly between host and device memory in order to be processed by the GPU [67, chapt. 5]. Therefore, *cudaMemcpy()* is used with the parameter *cudaMemcpyDeviceToHost* or *cudaMemcpyHostToDevice* indicating the direction of the data transfer [58, chapt. 3].

As already mentioned, a simplified version of the C++ code has been ported onto GPUs in the course of the *CSCS GPU Hackathon* [22]. During the event, which took place from the 20th to the 29th of september 2021, the supercomputer *Piz Daint* from the Swiss National Supercomputing Centre was used as main system [16]. Being an important perspective for the optimization of the solver, however, a full realization of GPU parallelism of the whole algorithm was not possible before the end of the project [22].

Here, we present an example of porting the computation of the residual to the GPU. Therefore, we add an extra device function specified with the *__global__*-qualifier for the parallel computations. Before calling the kernel with the triple-angle-brackets by the corresponding CPU function, all required arrays are allocated on the device, denoted by *_dev* (for device) to help differentiating between the different memory spaces. Then, the input data is copied to the GPU memory and the kernel is called to be executed simultaneously by $n_blocks \cdot n_threads$ threads in total. The objective hereby is to use as many threads as there are elements in the vectors we want to treat so that no iteration is required anymore within the GPU function and each thread only executes one single computation. In order to use enough threads, we define $n_threads = 32$ and calculate $n_blocks = ceil(m/n_threads)$ with m being the size of the vectors. Within the kernel, an identification number of each thread is determined via

$$id = blockIdx.x \cdot blockDim.x + threadIdx.x, \tag{82}$$

using the ID $blockIdx.x$ and dimension $blockDim.x$ of the current blocks as well as the ID of the thread $threadIdx.x$, followed by an execution of the corresponding computation. Since the indexing system of the grid is intended for 2-dimensional domains such as matrices or image processing, the addition '$.x$' is required for the identification within a simple 1D grid [58, chapt. 4]. After the kernel execution, the resulting data is copied back from the device to the host and the allocated GPU memory is released. The following code fragment presents the structure of the functions *gmgpolar::compute_residual* applied on the CPU, and *compute_residual_gpu* parallelised using CUDA as described above.

```
void gmgpolar::compute_residual(int l) {
    int m = v_level[l]->m;
    std::vector<double> Au(m, 0);
    v_level[l]->apply_A(v_level[l]->u, Au);
    v_level[l]->res = std::vector<double>(m);
    double *dev_Au, *dev_fVec, *dev_res;
    int n_threads = 32;
    int n_blocks = ceil(m / n_threads);

    cudaMalloc((void**) &dev_Au, m*sizeof(double));
    cudaMalloc((void**) &dev_fVec, m*sizeof(double));
    cudaMalloc((void**) &dev_res, m*sizeof(double));

    cudaMemcpy(dev_Au, &Au[0], m*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_fVec, &(v_level[l]->fVec[0]), m*sizeof(double),
               cudaMemcpyHostToDevice);
    cudaMemcpy(dev_res, &(v_level[l]->res[0]), m*sizeof(double),
               cudaMemcpyHostToDevice);

    compute_residual_gpu<<<n_blocks,n_threads>>>(m, dev_Au, dev_fVec, dev_res);

    cudaMemcpy(&(v_level[l]->res[0]), dev_res, m*sizeof(double),
               cudaMemcpyDeviceToHost);

    cudaFree(dev_Au);
    cudaFree(dev_fVec);
    cudaFree(dev_res);
}

__global__ void compute_residual_gpu(int m, double *Au, double *fVec, double *res) {
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    if (id < m){
        res[id] = fVec[id] - Au[id];
    }
}
```

Eventually, all functions of the multigrid algorithm have been ported to the GPU by the same principle, allocating all data on the device only once in the beginning. Since CUDA is based on standard C language, all C++-specific elements and the object oriented programming style have to be removed. Consequently, we had to convert all *std::vectors* to simple arrays, for example, and pass the variables directly to the functions as arguments instead of using class attributes. This, in fact, results in a separation of data from the numerics of the algorithm. In Table 17, the evolution of the runtime for the GPU parallelisation of a reduced version of the multigrid algorithm can be seen. As the runtime does not increase yet for the parallelised version, this also needs to be analysed further in future work.

$n_r \times n_\theta$	**CPU version**	**GPU version**	
		Computation time	Smoother/ Direct-Solver (non-optimised)
49×64	0.02	0.84	0.04
97×128	0.07	0.92	0.15
193×256	0.22	1.34	0.48

Table 17: Comparison of the runtime in seconds of the sequential CPU and the parallel GPU version.

6.5 Further optimizations

Having already discussed several approaches and ideas for the optimization of the multigird solver, what else could be done to go beyond the scope of this thesis?

Libraries

One possibility is the use of libraries either for the storage of sparse matrices, for the sparse matrix vector multiplications, or for the solving of linear matrix systems.

Further code optimizations

Vectorization and cache efficiency could be greatly improved in our solver. In particular, the next step would be to use monitoring tools in order to identify parts of the code as bottlenecks in terms of performance, and target these parts in code optimizations.

Improvement of the numerical method

Furthermore, there are several ideas for the modification of the mathematical discretizations and multigrid algorithm, such as an additional refinement in θ-direction on the outer radii of the geometries in order to obtain grid elements of about the same size.

Also, reconsidering again the treatment of the singularity at the origin, an even better approach may be found, using for example the origin as discretization node in combination with a special stencil which only takes the average of about ten naturally distributed points from the next radius instead of all neighbouring nodes.

The solving of tridiagonal linear systems of equations for the radial smoothers, or systems with periodic conditions for the circle smoothers, in the course of the multigrid cycle, could be improved by another more efficient solver than classical Gaussian elimination, e.g. Thomas algorithm [63].

Improvement of the implicit extrapolation

While currently using an uniform refinement between the two finest grid levels for the implicit extrapolation, the effect on dividing the intervals not exactly in the middle could be studied. However, in this case, also the prolongation operator P_{ex} needs to be modified.

Furthermore, an adaption of the whole multigrid cycle using successive steps of different multigrid iterations in order to improve the quality of the solution while reaching the solution much faster after less iterations could be analyzed. For example, we could have the following successive steps:

- a few (3,3)-cycles of standard multigrid,
- a few (1,1)-cycles of standard multigrid,
- a few (3,3)-cycles of multigrid with alternative extrapolation,
- a few (1,1)-cycles of multigrid with alternative extrapolation,
- one final (1,1)-cycle of multigrid with normal implicit extrapolation.

The numbers in the brackets hereby indicate the number of pre- and postsmoothing steps. This combination of distinct iterations is arranged in a way that we begin with a solver which delivers a low-order solution as fast as possible and then add some more time consuming steps that contribute to an improvement of the accuracy. Since already using a good approximation of the solution when starting the more expensive iterations, higher order is expected to be achieved much faster than without the prior steps. Using for instance (3,3)-cycles before (1,1)-cycles is the result of the fact that, the more smoothing we use, the faster the convergence but the poorer also the quality of the solution.

7 Conclusion

In this thesis, we have presented our solver GmgPolar for the two dimensional Poisson-like equation arising from the gyrokinetic code GyselaX for plasma simulations. Within the European project EoCoE, we implemented the matrix-free geometric multigrid solver with implicit extrapolation and combined zebra line smoothing in C++ achieving the same convergence and accuracy as with the already existing Matlab code. We hereby used line relaxation in order to overcome the anisotropy, arising due to the use of generalised polar coordinates, and combined the line splitting in circle and radial direction depending on the position within the domain. The application of the implicit extrapolation to the multigrid scheme increases the convergence order of the solution from two to up to four with only a small amount of additional computational work.

Furthermore, we optimized the code to increase the computational efficiency, focusing on the construction and application of large matrices. We reduced the overall number of operations by getting rid of naive implementations and redundant computations. Moreover, a decoupling of the circle and the radial smoothers was investigated in order to enable a simple parallelisation - but in fact, it turns out that this approach almost doubles the iteration count. Therefore, the *coupled circle-radial smoothing* emerges as superior among the two smoothing variants, requiring a future study of its suitability for parallelisation. Using full grid smoothing in combination with the implicit extrapolation reduces the number of iterations while decreasing the approximation order of the solution only slightly. However, the standard convergence criterion does not work anymore and thus, the error is used instead. Requiring the exact solution for the error computation in this case is an immense issue for the application to real world problems. Finally, the code has been investigated regarding its potential for parallelisation. Some parts of the code have already been parallelised using OpenMP or GPUs, but the complete and scalable parallelisation exceeds the scope of this thesis.

Therefore, future work will be required to fully optimise and parallelise the GmgPolar solver [22], before comparing it to other possible approaches (e.g. AMRex, or a splines solver) in terms of accuracy, convergence and cost throughout the overall project. Finally, the integration of the solvers into GyselaX is expected to result in drastic a reduction of the overall simulation costs.

To conclude, plasma fusion is an important recent topic within the context of the energy transition. However, being still far away from a commercial use of fusion power and without the definitive knowledge of the final success of all research, we might ask whether all those extremely expensive research projects are actually worth it. Not knowing whether nuclear fusion might ever be competitive with other energy sources at all, the political support and funding might only be an immense waste of money and a huge mistake. On the other hand, we might just have to give the research enough time to learn from errors and current failures, as well as focus on simulations instead of real-world experiments first. Nevertheless, knowing the immense potential and incredible possibilities of nuclear fusion, we have the responsibility as one of the richest continent of the world to investigate such options [42].

Finally, I would like to cite Harald Lesch at this point:

> "ITER heißt nicht nur *International Thermonuclear Experimental Reactor*; es heißt im Lateinischen auch der *Weg* - und der Weg ist das Ziel" [42].

In english: "ITER does not only stand for *International Thermonuclear Experimental Reactor*, but it also means *the journey* in Latin - and the journey is the destination"; German saying meaning that the efforts and procedures to achieve a goal are already worth it.

References

[1] Altfeld, H.-H. *Ein Blick auf Kernfusion und ITER*, Deutsche Physikalische Gesellschaft e. V., 2021, www.youtube.com/watch?v=5kjvHoYpLZQ.

[2] Amestoy, P. R.; Duff, I. S.; L'Excellent, J.-Y.; Koster, J. *A fully asynchronous multifrontal solver using distributed dynamic scheduling*, SIAM Journal on Matrix Analysis and Applications, 2001, vol. 23, no 1, p. 15-41.

[3] Barros, S. *The Poisson equation on the unit disk: a multigrid solver using polar coordinates*, Elsevier Inc, New York, 1988.

[4] Bergen, B. K.; Hülsemann, F. *Hierarchical hybrid grids: data structures and core algorithms for multigrid.* Numerical linear algebra with applications, 2004, 11(2-3), 279-291.

[5] Bernert, K. *Tau-extrapolation - theoretical foundation, numerical experiment, and application to navier–stokes equations*, Siam Publications, Philadelphia, 1997.

[6] Bernert, K.; Jung, M.; Rüde, U. *Multigrid Tau-Extrapolation for Nonlinear Partial Differential Equations*, ICOSAHOM 95, 1996.

[7] Börm, S.; Hiptmair, R. *Analysis of tensor product multigrid*, Numerical Algorithms 26, Springer, 2001.

[8] Bouzat, N.; Bressan, C.; Grandgirard, V. et al. *Targeting realistic geometry in Tokamak code Gysela*, EDP Sciences, 2018.

[9] Brandt, A. *Multigrid Techniques: 1984 guide with applications to fluid dynamics*, St. Augustin, 1984.

[10] Briggs, W. L.; Henson, Van E. *A Multigrid Tutorial, part 1 and 2*, presentation.

[11] Briggs, W. L.; Henson, Van E.; McCormick, S. F. *A Multigrid Tutorial*, Siam publications, 2000.

[12] Büchner, J. *Vlasov code simulation. Advanced Methods for Space Simulations*, 23-46, 2007.

[13] Chandra, R.; Dagum, L.; et. al. *Parallel Programming in OpenMP*, Academic Press, 2001.

[14] Cheng, J.; Grossman, M.; McKercher, T. *Professional Cuda C Programming*, John Wiley & Sons, 2014.

[15] Clixoom Science & Fiction. *Durchbruch: Rekord-Kernfusion in China erreicht*, United Creators PMB GmbH, Berlin, 2020, www.youtube.com/watch?v=GF2ikwq-_V0.

[16] CSCS - Swiss National Supercomputing Centre. *Piz Daint*, 2021, www.cscs.ch/computers/piz-daint/.

[17] De Clercq, G. *Nuclear fusion reactor ITER's construction accelerates as cost estimate swells*, London, Reuters, 2016, www.reuters.com/article/us-france-nuclear-iter-idUSKCN1271BC.

[18] Dongarra, J. *Data Structures*, Netlib Repository at UTK and ORNL, 2020, www.netlib.org/linalg/html_templates/node89.html.

[19] Duff, I. S.; Erisman, A. M.; Reid, J. K. *Direct methods for sparse matrices*, Oxford University Press, 2017.

[20] Economist. *Stellar work*, 2015, www.fusion4freedom.com/stellar-work/.

[21] EoCoE-II *D3.1 Co-design of LA solvers, Specification of characteristics and interfaces of the LA solvers for all target applications*, European Commission, INFRAEDI-824158, 2019.

[22] EoCoE-II *D3.3 Updated results and new releases of LA solvers*, European Commission, INFRAEDI-824158, 2020.

[23] EoCoE official website, https://www.eocoe.eu/.

[24] Eurofusion homepage, www.euro-fusion.org/.

[25] Evers, M. *Kernspaltung*, Physikunterricht-online, 2021, www.physikunterricht-online.de/jahrgang-12/kernspaltung/.

[26] Grandgirard, V. et al. *A 5d gyrokinetic full-f global semi-lagrangian code for flux-driven ion turbulence simulations*, Elsevier B.V, 2016.

[27] Greiner, G. *Algorithmik kontinuierlicher Systeme: Matrizen - Datenstruckturen und praktische Verfahren*, Friedrich-Alexander-Universität, lecture slides, 2016.

[28] Hackbusch, W. *Multigrid methods and applications: with 48 tab.*, Springer, Berlin, 1985.

[29] Hoffmann, S.; Lienhart, R. *OpenMP - Eine Einführung in die parallele Programmierung mit C/C++*, Springer-Verlag, 2008.

[30] Iter organization, 2020, www.iter.org/.

[31] Iyengar, S. R. K., Goyal, A. *A note on multigrid for the three-dimensional Poisson equation in cylindrical coordinates*, Elsevier B.V, Amsterdam, 1990.

[32] Joachim Herz Stiftung. *Kernspaltung und Kernfusion*, Leifiphysik, 2021, www.leifiphysik.de/kern-teilchenphysik/kernspaltung-und-kernfusion/grundwissen/kernspaltung.

[33] Jung, M.; Rüde, U. *Implicit extrapolation methods for multilevel finite element computations*, Society for Industrial and Applied Mathematics, Philadelphia, 1996.

[34] Jung, M.; Rüde, U. *Implicit extrapolation methods for variable coefficient problems*, Siam Publications, Philadelphia, 1998.

[35] Kang, K. S., Kormann, K., Grandgirard, V. *Solving Poisson's equation based on AMReX*, Max-Planck-Institute for plasma physics, 2021.

[36] Kühn, M. J.; Kruse, C.; Leleux, P.; Rüde, U. *Gmgpolar/ iexmg: complexity analysis*, unpublished yet.

[37] Kühn, M. J.; Kruse, C.; Leleux, P. et al. *GMGpolar/iexmg: implementation in C++*, presentation, 2021.

[38] Kühn, M. J.; Kruse, C.; Rüde, U. *Energy-Minimizing, Symmetric Discretizations for Anisotropic Meshes and Energy Functional Extrapolation*, SIAM J. Sci. Comput. Vol. 43(4), pp. A2448-A2473 (2021).

[39] Kühn, M. J.; Kruse, C.; Rüde, U. *Implicitly extrapolated geometric multigrid on disk-like domains for the gyrokinetic Poisson equation from fusion plasma applications*, Preprint: https://hal.archives-ouvertes.fr/hal-03003307/, Submitted to Journal of Scientific Computing, 2021.

[40] Latu, G.; Grandgirard, V. et al. *Improving conservation properties of a 5D gyrokinetic semi-Lagrangian code*, Springer, Berlin/Heidelberg, 2014.

[41] Lee, W. W. *Gyrokinetic approach in particle simulation*, Physics of Fluids 26 (2), 556-562, 1983.

[42] Lesch, H.. *Kernfusion: Klimaretter oder Milliardengrab?*, ZDF Terra X Lesch & Co, 2020, www.youtube.com/watch?v=nVTcirxdRWM.

[43] Lui, S. H. *Numerical Analysis of Partial Differential Equations*, John Wiley & Sons Inc., 2011.

[44] Masturah, N. et al. *On comparison of multigrid cycles for poisson solver in polar plane coordinates*, IEEE, 2015.

[45] McCormick, S. F.; Rüde, U. *On Local Refinement Higher Order Methods for Elliptic Partial Differential Equations*, International Journal of High Speed Computing, 1990.

[46] MUMPS. *MUMPS : a parallel sparse direct solver*, 2021, http://mumps.enseeiht.fr/.

[47] National Institutes for Quantum Science Technology. *JT-60U Experimental Report - JT-60U Reaches 1.25 of Equivalent Fusion Power Gain*, 2018, www.qst.go.jp/site/jt60-english/5593.html.

[48] NTT. *Conclusion of a Cooperation Agreement with ITER*, businesswire, 2020, www.businesswire.com/news/home/20200514005951/en/NTT-Conclusion-of-a-Cooperation-Agreement-with-ITER.

[49] Olson, L. *Multigrid Methods*, Copper Multigrid Conference, Illinois, 2021.

[50] OpenMP Architecture Review Board. *OpenMP API specification - depend clause*, 2018, www.openmp.org/spec-html/5.0/openmpsu99.html.

[51] Quinn, M. J. *Parallel Programming - in C with MPI and OpenMP*, McGraw-Hill, 2003.

[52] Rüde, U. *Extrapolation and related techniques for solving elliptic equations*, TU München, 1991.

[53] Rüde, U. *Extrapolation techniques for constructing higher order finite element methods*, TU München, 1993.

[54] Rüde, U. *Mehrgittermethode*, Springer Nature B.V, Heidelberg, 2019.

[55] Rüde, U. *Multiple tau extrapolation for multigrid methods*, München, 1987.

[56] Rüde, U. *The hierarchical basis extrapolation method*, Society for Industrial and Applied Mathematics, 1992.

[57] Rüde, U., Private communications, 05.08.2021, 08.09.2021, and 06.12.2021.

[58] Sanders, J.; Kandrot, E. *Cuda by example - An Introduction to General-Purpose GPU Programming*, Nvidia Corporation, Addison-Wesley, 2011.

[59] Shapira, Y. *Matrix-based Multigrid: Theory and Applications*, Springer Science+Business Media, LCC, 2008.

[60] Sonnendrücker, E., Private discussion, 18.05.2021.

[61] Stacey, W. M. *Fusion Plasma Physics*, Wiley-VCH, Weinheim, 2005.

[62] Stüben, K.; Trottenberg, U. *Multigrid methods: Fundamental algorithms, model problem analysis and applications*, in Multigrid methods, Springer, 1982.

[63] Trefethen, L. N.; Bau, D. *Numerical Linear Algebra*, Siam, 1997.

[64] Trottenberg, U.; Oosterlee, C.; Schüller, A. *Multigrid*, Acad. Press, San Diego, 2001.

[65] Wesseling, P. *An introduction to multigrid methods*, John Wiley & Sons, England, 1992.

[66] Wesson, J. *Tokamaks*, Clarendon Press, Oxford, 1997.

[67] Wilt, N. *The Cuda Handbook - A Comprehensive Guide to GPU Programming*, Pearson Education, Addison-Wesley, 2013.

[68] Zhang, W. et al. *Amrex : a framework for block-structured adaptive mesh refinement*, The Journal of Open SOurce Software, 2019.

[69] Zoni, E. *Theoretical and numerical studies of gyrokinetic models for shaped Tokamak plasmas*, TU München, 2019.

[70] Zoni, E.; Güçlü, Y. *Solving hyperbolic-elliptic problems on singular mapped disk-like domains with the method of characteristics and spline finite elements*, 2019.

[71] Zoni, E., Güçlü, Y., Sonnendrücker, E. *Solving the guiding-center model on singular mapped disk-like domains*, Garching, 2018.