Complexity analysis and scalability of a matrix-free extrapolated geometric multigrid solver for curvilinear coordinates representations from fusion plasma applications

Philippe Leleux, Christina Schwarz, Martin Joachim Kühn, Carola Kruse, Ulrich Rüde

Noname manuscript No.
(will be inserted by the editor)

Complexity analysis and scalability of a matrix-free extrapolated geometric multigrid solver for curvilinear coordinates representations from fusion plasma applications

Philippe Leleux · Christina Schwarz · Martin J. Kühn · Carola Kruse · Ulrich Rüde

Received: date / Accepted: date

Abstract Tokamak fusion reactors are promising alternatives for future energy production. Gyrokinetic simulations are important tools to understand physical processes inside tokamaks and to improve the design of future plants. In gyrokinetic codes such as Gysela, these simulations involve at each time step the solution of a gyrokinetic Poisson equation defined on disk-like cross sections. The authors of [KKR21,KKR22] proposed to discretize a simplified differential equation using symmetric finite differences derived from the resulting energy functional and to use an implicitly extrapolated geometric multigrid scheme tailored to problems in curvilinear coordinates. In this article, we extend the discretization to a more realistic partial differential equation and demonstrate the optimal linear complexity of the proposed solver, in terms of computation and memory. We provide a general framework to analyze flops and memory usage of matrix-free approaches for stencil-based operators. Finally, we give an efficient matrix-free implementation fo the considered solver exploiting a task-based multithreaded parallelism which takes advantage of the disk-shaped geometry of the problem. We demonstrate the parallel efficiency for the solution of problems of size up to 50 million unknowns.

Keywords Multigrid · complexity · curvilinear coordinates · parallelization · multithreading · Plasma fusion

Philippe Leleux (Corresponding author)
Laboratoire d'Analyse et d'architecture des Systèmes (LAAS), équipe TSF, 7, avenue du Colonel Roche BP 54200, 31031 Toulouse cedex 4, France. ORC-ID: 0000-0002-3760-4698
E-mail: pleleux@laas.fr

Martin J. Kühn
German Aerospace Center (DLR), Institute for Software Technology, Department for High-Performance Computing, Cologne, Germany. ORC-ID: 0000-0002-0906-6984
E-mail: Martin.Kuehn@DLR.de

Christina Schwarz
University of Bayreuth, Universitätsstr. 30, 95447 Bayreuth, Germany
E-mail: christina.schwarz@uni-bayreuth.de

Carola Kruse
Parallel Algorithms Team, CERFACS (Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique), 42 Avenue Gaspard Coriolis, 31057 Toulouse Cedex 01, France. ORC-ID: 0000-0002-4142-7356
E-mail: carola.kruse@cerfacs.fr

Ulrich Rüde
Friedrich-Alexander Universität Erlangen-Nürnberg, Cauerstr. 11, 91058 Erlangen, Germany. ORC-ID: 0000-0001-8796-8599 E-mail: ulrich.ruede@fau.de

Mathematics Subject Classification (2020) 68Q25 · 65Y20 · 65Y05 · 65N55 · 65N06 · 65B99

1 Introduction

With accelerating climate change, plasma fusion is a promising alternative for future energy productions. Tokamak fusion reactors are actively studied objects to realize energy production from plasma fusion; see, e.g., [WC11] and the references therein. In these torus-shaped reactors, plasma is magnetically confined to obtain energy from physical processes. Empirical laws are important to design current reactors such as ITER or JET[1]. With the advent of modern supercomputers, simulation and high performance computing (HPC) have opened up new possibilities to study potential reactor designs. Simulation codes can lead to improved designs by creating a better understanding for physical aspects such as the transport of plasma. However, the simulation of plasma in these reactors still involves high cost in terms of computation and memory, and thus requires the use of efficient codes. The *gyrokinetic* framework is of particular interest in this context since it is able to capture the turbulence in the plasma, and allows to decrease the dimensions of the system from 6 to 5 dimensions: three for the torus geometry and two for the velocity [BBG+18, GAB+16]. In gyrokinetic codes, a 5D Vlasov equation coupled with a 3D Poisson-like equation must be solved at each time step. While being of smaller dimension, the solution of the latter 3D system is still computationally expensive and its compute time as well as the needed resources are non negligible. In current implementations such as Gysela [GAB+16] or Jorek [HHP+21], the 3D equation is reduced to the solution of 2D Poisson-like equations on poloidal cross-sections of the tokamak. The description of the domain by curvilinear coordinates [BBG+18, ZG19, Zon19], and the consideration of a varying density field [ZG19, Zon19], lead to anisotropic operators and nonuniform meshes. In this case, a standard derivation of a finite difference discretization may lead to a non-symmetric linear system, although the corresponding energy functional is symmetric. In [KKR21], symmetric finite difference schemes based on minimizing the energy functional were derived. A rigorous analysis showed that, when using nonstandard finite element discretizations, extrapolation formulas [JR96] applied on the discrete energy functional lead to higher order approximations. Numerical results showed that additionally, the likewise extrapolated finite difference schemes lead to an order of convergence of four.

Multigrid solvers are among the fastest numerical methods to solve elliptic partial differential equations and are often optimal in the sense that the number of arithmetic operations is proportional to the degrees of freedom of the discretized problem [TOS01]. Multigrid algorithms can be divided into the class of algebraic and geometric algorithms. The focus of this paper is on a tailored geometric multigrid solver for a Poisson-like equation on disk-like domains. The extrapolation techniques presented in [KKR21] can be incorporated naturally in a multigrid scheme, as already shown by [JR96, JR98]. In [KKR22], this implicitly extrapolated geometric multigrid method based on symmetric finite difference schemes, and named GMGPolar, was considered for a simplified version of the gyrokinetic Poisson equation. The algorithm was shown to deliver a higher order convergence in only a small number of solver iterations.

When it comes to computational expense, two different foci are memory and computation. Algorithms are called *memory-bound* if their execution speed is limited by the speed of data transfer (between different levels of memory). Algorithms are called *compute-bound* if their execution speed is limited by the speed of the computational units, e.g., the cores of the computing

[1] https://www.iter.org/fr/sci/beyonditer

node. For distributed-memory-based applications also the network speed can become a limiting factor.

In the present article, a more realistic version of the Poisson-like equation from [KKR22] is considered. We provide an efficient C/C++ implementation of GMGPolar that is available open-source. The published code can be found GitHub under a permissive Apache-2.0 license. Its release version 1.0 [LKS+23] has been used for this article. We demonstrate optimal linear complexity of the solver with a systematic analysis of computational and memory costs. The algorithm employs a task-based multithreaded parallelism which takes advantage of the disk-shaped geometry of the problem. Such parallel implementation is essential for large scale simulations of plasma in tokamak reactors. In [BLK+23], it has recently been shown that GMGPolar is able to compete with other state-of-the-art solvers for the considered applications. Our rigorous computation and memory analysis is built in the sense that it can be transferred to other stencil-based operators.

The article is structured as follows. In Section 2, we introduce the model problem, and in Section 3 we introduce the discretization used for the model problem and geometry. In Section 4, we briefly rephrase the implicitly extrapolated geometric multigrid solver, then in Section 5 we provide a detailed complexity analysis. The corresponding details on the particular function evaluations and our rigorous framework for stencil-based operators is to be found in the Appendix. The parallelization is discussed in Section 6 and numerical results can be found in Section 7.

2 Model problem

We are interested in the solution of the Poisson-like equation

$$\begin{aligned} -\nabla \cdot (\alpha \nabla u) + \beta u &= f \quad \text{in} \quad \Omega, \\ u &= u_D \quad \text{on} \quad \partial\Omega, \end{aligned} \tag{1}$$

where $\Omega \subset \mathbb{R}^2$ is a disk-like domain, $f : \Omega \to \mathbb{R}$, $f \in C^0(\overline{\Omega})$, is the right hand side, $\alpha, \beta : \Omega \to \mathbb{R}$ with $\alpha \in C^1(\Omega) \cap C^0(\overline{\Omega})$ and $\beta \in C(\overline{\Omega})$ are coefficients corresponding to *density profiles*. Furthermore, we have the Dirichlet boundary conditions $u_D \in C^0(\overline{\partial\Omega})$. For more details on this equation, see [Zon19, eq. (4.36)]. In [KKR22], Equation (1) was considered with $\beta = 0$. Particular choices for the coefficients α and β will be presented for numerical experiments in Section 7.

As the discretization method presented in the next section will be based on the energy functional corresponding to the partial differential equation (1), we also briefly introduce this energy functional. The problem (1) can be traced back to the minimization of the energy functional

$$J(u) := \int_\Omega \frac{1}{2}\alpha |\nabla u|^2 + \frac{1}{2}\beta u^2 - f u \, \mathrm{d}(x, y), \tag{2}$$

over a suitable Sobolev space incorporating the boundary conditions u_D.

As in [BBG+18, ZG19, Zon19], we describe our domains Ω by curvilinear coordinates, i.e. based on a mapping F from the Cartesian coordinates (x, y) to the (generalized) polar coordinates or curvilinear coordinates $(r, \theta) \in [R_{\min}, R_{\max}] \times [0, 2\pi)$ where r is the (generalized) radius and θ the (generalized) angle. Figure 1 shows an example of such coordinates and mapping. This description introduces a periodicity constraint in the θ-direction. Figure 1 provides an example of a circular domain described by a grid in Cartesian and polar coordinates. Please note that the mapping between both systems is only bijective if $R_{\min} > 0$. In [KKR22], we have considered different approaches to handle the area around the origin. Similarly to [GAB+16, KKR22], we consider that $R_{\min} \approx 0$, e.g. $R_{\min} = 10^{-6}$, and prescribe (homogeneous) Dirichlet boundary conditions on the artificial inner boundary.

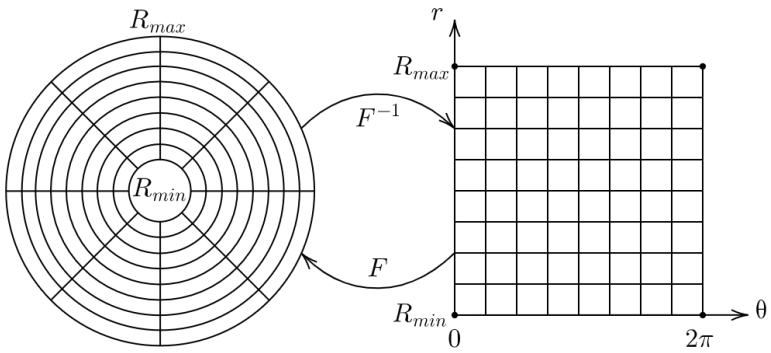


Fig. 1: A circular geometry described *(Left)* in Cartesian coordinates and *(Right)* in polar coordinates $(r, \theta) \in [R_{min}, R_{max}] \times [0, 2\pi)$ with mappings F and F^{-1} between the different descriptions.

As presented in Figure 1, we describe a circular geometry by simple polar coordinates and an invertible mapping F from $\widetilde{\Omega} := [R_{min}, R_{max}] \times [0, 2\pi)$ onto the circle. In order to model more realistic cross-sections, generalized polar or curvilinear coordinates need to be used. Particular geometries have been introduced in [BBG+18, CH08, ZG19]. We now introduce two mappings F_S and F_C which describe geometries by $F_S(\widetilde{\Omega})$ and $F_C(\widetilde{\Omega})$, see Figure 2.

The *Shafranov geometry* is a deformed ellipse defined by the mapping

$$F_S(r, \theta) := \begin{pmatrix} x(r, \theta) \\ y(r, \theta) \end{pmatrix} = \begin{pmatrix} x_0 + (1 - \kappa) r \cos\theta - \delta r^2 \\ y_0 + (1 + \kappa) r \sin\theta \end{pmatrix}, \tag{3}$$

where κ is the elongation and δ is the Shafranov shift, see [BBG+18, ZG19]. In [KKR21, KKR22], this geometry was simply denoted *deformed geometry*, and $x_0 = y_0 = 0$ were removed from the equation.

The *Czarny geometry* adds triangularity to the shape with the mapping

$$F_C(r, \theta) := \begin{pmatrix} x(r, \theta) \\ y(r, \theta) \end{pmatrix} = \begin{pmatrix} \frac{1}{\varepsilon}\left(1 - \sqrt{1 + \varepsilon\,(\varepsilon + 2\, r\, \cos\theta)}\right) \\ y_0 + \frac{\lambda\, \xi\, r\, \sin\theta}{2 - \sqrt{1 + \varepsilon\,(\varepsilon + 2\, r\, \cos\theta)}} \end{pmatrix}, \tag{4}$$

where ε is the inverse aspect ratio, λ the ellipticity, and $\xi = 1/\sqrt{1 - \varepsilon^2/4}$, see [CH08, ZG19].

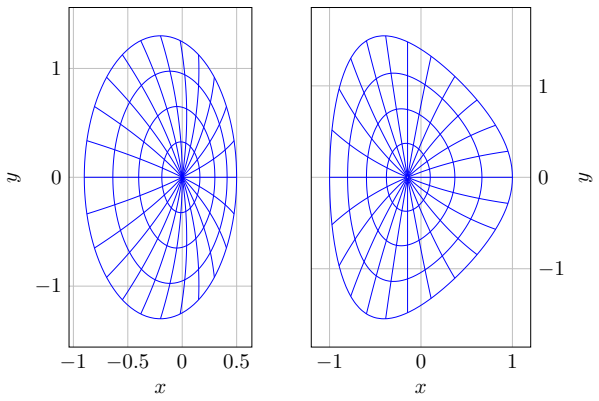
In Figure 2, we present the Shafranov geometry for $x_0 = y_0 = 0$, $\kappa = 0.3$, and $\delta = 0.2$ as well as the Czarny geometry for $y_0 = 0$, $\varepsilon = 0.3$, and $\lambda = 1.4$.

3 Discretization

In the following, we introduce the mesh as well as the finite difference discretization with respect to the logical domain $\widetilde{\Omega} := [R_{min}, R_{max}] \times [0, 2\pi)$.

The mesh Ω_1 will be in product (i.e., tensor) format and given by two separate subdivisions $r_1, \ldots, r_{n_r} \in [R_{\min}, R_{\max}]$ and $\theta_1, \ldots, \theta_{n_\theta+1} \in [0, 2\pi]$. These subdivisions are arbitrary with definitions

$$\begin{gathered} r_1 = R_{\min}, \quad r_{n_r} = R_{\max}, \quad \theta_1 = 0, \quad \text{and} \quad \theta_{n_\theta+1} = 2\pi; \\ r_{i+1} = r_i + h_i, \quad i \in \{1, \ldots, n_r - 1\}, \quad \theta_{j+1} := \theta_j + k_j, \quad j \in \{1, \ldots, n_\theta\}, \end{gathered} \tag{5}$$



(a) Shafranov geometry with $x_0 = y_0 = 0$, $\kappa = 0.3$, and $\delta = 0.2$

(b) Czarny geometry with $y_0 = 0$, $\varepsilon = 0.3$, and $e = 1.4$

Fig. 2: Two geometries are considered.

and assumptions

$$\begin{aligned} n_r \text{ odd}, \quad & h_{2i} = h_{2i-1}, \qquad i \in \{1, \dots, (n_r - 1)/2\}, \\ n_\theta \text{ even}, \quad & k_{2j} = k_{2j-1}, \qquad j \in \{1, \dots, n_\theta/2\}. \end{aligned} \tag{6}$$

Remark 1 The assumptions in Equation (6) mean that there is a specific structure needed on pairs of rectangular elements. The assumption (6) is needed to prove the relations between quadratic and extrapolated linear finite elements in [KKR21] which increase the order of convergence of the resulting multigrid scheme. There is no proof for finite differences, but it was shown numerically on these grids that they behave similarly as their finite element counterpart. Note that this assumption is not needed if the discretization is used for geometric multigrid without the proposed extrapolation step. For more details, see [KKR21].

We further assume h_i and k_j to be uniformly bounded by

$$0 < h_{\min} \le h_i \le h \quad \text{and} \quad 0 < k_{\min} \le k_j \le k, \tag{7}$$

as well as the existence of $0 < \tau < \infty$ such that $h = \tau k$. Our finite difference discretization builds upon *the localization* (11) *of the energy functional* (2) first, *the discretization of this localized form* second, and *the differentiation of the sum of the localized, discretized energies* third.

In order to localize Equation (2), consider any rectangular element $R_{ij} := [r_i, r_i+h_i]\times[\theta_j, \theta_j+k_j]$ of the logical domain $\widetilde{\Omega}$. For $R_{i,j}$, we have its curvilinear representation by $F(R_{ij})$. Here, F typically corresponds to the Shafranov or Czarny geometry but could be given by another invertible mapping.

The following paragraphs are essentially built upon the theory in [KKR21] and only differ due to the coefficients α and β. In [KKR21], we allowed for a more general diffusion tensor $A : \mathbb{R}^2 \to \mathbb{R}^2$ instead of $\alpha : \Omega \to \mathbb{R}$ which could, in principle, also be used here. Also, in [KKR21], we used $\beta = 0$. In order to make the paper self-contained, we repeat some of the basic ideas.

To make the transformation between Cartesian and curvilinear coordinates clear, let us use

$$\widetilde{\alpha}(x, y) = \alpha(r, \theta), \quad \widetilde{\beta}(x, y) = \beta(r, \theta), \quad \widetilde{f}(x, y) = f(r, \theta), \quad \widetilde{u}(x, y) = u(r, \theta). \tag{8}$$

By transformation of the energy functional in Equation (2), we then obtain

$$\begin{aligned} J_{R_{i,j}}(u) :&= \int_{F(R_{i,j})} \left(\frac{1}{2}\alpha |\nabla_{(x,y)} \widetilde{u}|^2 + \frac{1}{2}\widetilde{\beta}\widetilde{u}^2 - \widetilde{f}\widetilde{u} \right) \mathrm{d}(x,y) \\ &= \int_{R_{i,j}} \left(\frac{1}{2}\alpha |DF^{-T} \nabla_{(r,\theta)} u|^2 + \frac{1}{2}\beta u^2 - fu \right) |\det DF| \, \mathrm{d}(r,\theta), \end{aligned} \tag{9}$$

where DF is the Jacobian matrix of F and $DF^{-T} := (DF^T)^{-1}$.

In principle, we have to distinguish the functions in Equation (8) as well as the operators with respect to the different coordinates throughout the paper but we will generally write u or ∇u and assume that the variables are clear from the context.

In order to simplify the notation, we define

$$\frac{1}{2}\alpha DF^{-1} DF^{-T} |\det DF| =: \begin{pmatrix} a^{rr} & \frac{1}{2}a^{r\theta} \\ \frac{1}{2}a^{\theta r} & a^{\theta\theta} \end{pmatrix}. \tag{10}$$

Since (10) is symmetric, we have $a^{r\theta} = \frac{1}{2}a^{r\theta} + \frac{1}{2}a^{\theta r}$.

We then have

$$J_{R_{i,j}}(u) = \int_{R_{i,j}} \left(a^{rr} u_r^2 + a^{r\theta} u_r u_\theta + a^{\theta\theta} u_\theta^2 + \left(\frac{1}{2}\beta u^2 - fu \right) |\det DF| \right) \mathrm{d}(r,\theta), \tag{11}$$

and the global energy can be decomposed as

$$J(u) = \sum_{i=1}^{n_r - 1} \sum_{j=1}^{n_\theta} J_{R_{i,j}}(u). \tag{12}$$

The symmetric finite difference stencils are now derived by discretizing $J_{R_{i,j}}(u)$. Here and in the following, we use for any function $v : \widetilde{\Omega} \to \mathbb{R}$ the notation

$$v_{i,j} := v(r_i, \theta_j). \tag{13}$$

The discretization of

$$\int_{R_{i,j}} \left(a^{rr} u_r^2 + a^{r\theta} u_r u_\theta + a^{\theta\theta} u_\theta^2 - fu |\det DF| \right) \mathrm{d}(r,\theta) \tag{14}$$

can be obtained by [KKR21, (3.15) and (3.8)]. We only need to consider the remaining part

$$\int_{R_{i,j}} \frac{1}{2}\beta u^2 |\det DF| \, \mathrm{d}(r,\theta). \tag{15}$$

Defining $b := \frac{1}{2}\beta u^2 |\det DF|$, the discretization of Equation (15) can be obtained by using the trapezoidal rule:

$$\int_{R_{i,j}} b \, \mathrm{d}(r,\theta) = \frac{h_i k_j}{4} (b_{i+1,j+1} + b_{i+1,j} + b_{i,j+1} + b_{i,j}) + O(h_i^3 k_j + h_i k_j^3). \tag{16}$$

Remark 2 Alternatively, we could use the midpoint rule and operators (3.3) and (3.4) of [KKR21] to obtain the same result.

The discretization (16) as well as the discretizations of the two summands in Equation (14) are linear and quadratic expressions in $u := (u_{1,1}, \dots, u_{n_r, n_{\theta+1}})$ with a local discretization error of $O(h_i^3 k_j + h_i k_j^3)$ (see [KKR21, (3.15) and (3.8)]). The sum over all elements then implicitly yields a discrete energy operator E as well as a vector $\widetilde{f}$ for $f|\det DF|$ such that

$$J(u) = \sum_{i=1}^{n_r-1} \sum_{j=1}^{n_\theta} \widetilde{J}_{R_{i,j}}(u) + O(h^2 + k^2) = \frac{1}{2} u^T E u - u^T \widetilde{f} + O(h^2 + k^2), \tag{17}$$

where $\widetilde{J}_{R_{i,j}}(u)$ stands for the discretization of $J_{R_{i,j}}(u)$ in Equation (11).

Instead of searching the minimum of the continuous energy functional $J(u)$, we now compute the derivative of the discretized formulation $\frac{1}{2} u^T E u - u^T \widetilde{f}$ with respect to the nodal values of u and set it to zero, i.e.,

$$\frac{\partial}{\partial u_{s,t}} \left(\frac{1}{2} u^T E u - u^T \widetilde{f} \right) = \sum_{i=1}^{n_r-1} \sum_{j=1}^{n_\theta} \frac{\partial}{\partial u_{s,t}} \widetilde{J}_{R_{i,j}}(u) \stackrel{!}{=} 0. \tag{18}$$

Due to the difference, averaging, and integration formulas used for discretization, we have

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

for $(i,j) \notin I_{s,t} := \{(s,t), (s-1,t), (s,t-1), (s-1,t-1)\}$.

Computing the derivatives for $(i,j) \in I_{s,t}$ and solving Equation (18) yields the nine point stencil with the horizontal and vertical connections

$$\begin{aligned}
u_{s+1,t} : (*9)_{s+1,t} &= -\frac{k_t + k_{t-1}}{h_s} \frac{a^{rr}(s,t) + a^{rr}(s+1,t)}{2}, \\
u_{s-1,t} : (*9)_{s-1,t} &= -\frac{k_t + k_{t-1}}{h_{s-1}} \frac{a^{rr}(s-1,t) + a^{rr}(s,t)}{2}, \\
u_{s,t+1} : (*9)_{s,t+1} &= -\frac{h_s + h_{s-1}}{k_t} \frac{a^{\theta\theta}(s,t) + a^{\theta\theta}(s,t+1)}{2}, \\
u_{s,t-1} : (*9)_{s,t-1} &= -\frac{h_s + h_{s-1}}{k_{t-1}} \frac{a^{\theta\theta}(s,t-1) + a^{\theta\theta}(s,t)}{2},
\end{aligned} \tag{19}$$

the central update

$$\begin{aligned}
u_{s,t} : (*9)_{s,t} = &- \left[(*9)_{s+1,t} + (*9)_{s-1,t} + (*9)_{s,t+1} + (*9)_{s,t-1} \right] \\
&+ \frac{(h_s + h_{s-1})(k_t + k_{t-1})}{4} \beta_{s,t} |\det DF_{s,t}|,
\end{aligned} \tag{20}$$

and the diagonal connections

$$\begin{aligned}
u_{s+1,t+1} : (*9)_{s+1,t+1} &= \frac{a^{r\theta}(s+1,t) + a^{r\theta}(s,t+1)}{4}, \\
u_{s+1,t-1} : (*9)_{s+1,t-1} &= \frac{a^{r\theta}(s+1,t) + a^{r\theta}(s,t-1)}{4}, \\
u_{s-1,t+1} : (*9)_{s-1,t+1} &= \frac{a^{r\theta}(s-1,t) + a^{r\theta}(s,t+1)}{4}, \\
u_{s-1,t-1} : (*9)_{s-1,t-1} &= \frac{a^{r\theta}(s-1,t) + a^{r\theta}(s,t-1)}{4}.
\end{aligned} \tag{21}$$

Finally, the right hand side is given by

$$\frac{(h_s + h_{s-1})(k_t + k_{t-1})}{4} f_{s,t} |\det DF_{s,t}|. \tag{22}$$

4 Geometric multigrid solver

Multigrid solvers are powerful numerical methods for solving partial differential equations; see, e.g., [TOS01]. In our paper, we will focus on a tailored geometric multigrid solver for Equation (1). In [KKR22], such a solver was proposed using two ingredients: a suited smoothing procedure for curvilinear discretizations, and an implicit extrapolation scheme. The smoothing procedure is motivated by [Bar88] while the implicit extrapolation scheme was introduced in [JR96, JR98]. The resulting solver is named *GMGPolar*.

To make this paper self-contained, we briefly introduce the solver used in the following numerical simulations. For more details on the derivation of the developed multigrid scheme, see [KKR22].

4.1 Multigrid hierarchy

In order to build a multigrid scheme, we need to construct a hierarchy of grids. Based on Equation (5), the finest grid is defined as a rectangular mesh Ω_1 of the logical domain, see also Figure 1. We then use standard coarsening to obtain coarser meshes. Basically, starting from the first node, we take every second value in $\{r_1, \dots, r_{n_r}\}$ and $\{\theta_1, \dots, \theta_{n_\theta+1}\}$ to build the second grid level Ω_2, and so on and so forth. Note that for all levels, the last values are naturally kept in each direction (r_{n_r} and $\theta_{n_\theta+1}$).

Assuming that there exist constants $C_r, C_\theta \in \mathbb{R}$ such that $n_r = C_r 2^L + 1$ and $n_\theta = C_\theta 2^L$ for $L \in \mathbb{N}$, we can construct a hierarchy of L grids $\Omega_1, \dots, \Omega_L$ from finest to coarsest. On each level $l \in \{1, \dots, L\}$, we then use the discretizations (19)–(22) to construct the system

$$A_l u_l = f_l, \tag{23}$$

where A_l is a matrix of size $m_l \times m_l$ and u_l, f_l are vectors of size m_l. On level l, with n_{rl} and $n_{\theta l}+1$ the number of divisions in the r and θ directions, we have $n_{rl} \times (n_{\theta l} + 1)$ nodes in the grid. In the θ-direction, we add the periodicity condition $\theta_1 = \theta_{n_{\theta+1}}$ and reduce the size of the matrix A_l to $m_l = n_r \cdot n_\theta$. Since standard coarsening is used on the 2D grid, we have that

$$4^{(l-1)} m_l \approx m_1. \tag{24}$$

The levels are linked by two transfer operators, the prolongation operator P_{l+1}^l transferring the information from the coarser level $l+1$ to the finer level l, and the restriction operator R_l^{l+1}, which is often defined by $R_l^{l+1} := P_{l+1}^{l\,T}$. The latter definition ensures that the operator

$$P_{l+1}^l A_{l+1} R_l^{l+1} \tag{25}$$

is symmetric when A_l is symmetric.

In [KKR22], the prolongation is defined as the bilinear interpolation operator on an anisotropic mesh. In Table 1, we provide an illustration of the stencil used for the interpolation P_{l+1}^l. For more details, see Section 4.3. Except for the implicit extrapolation, which will act only between the finest two levels, these stencils are qualitatively identical on any level, only depending on the prolongated node. Quantitatively, the values of the operators depend on the distance to the neighboring nodes.

In order to avoid a proliferation of indices, we skip the index l for a particular grid where our remarks hold for any level of the hierarchical grid $\Omega_1, \dots, \Omega_L$.

Prolongation	**Stencil**				**Reference**
r ↑, → θ	Coarse	Vertical	Horizontal	Diagonal	
central node (r_i, θ_j)	i odd, j odd	i even, j odd	i odd, j even	i even, j even	
P^l_{l+1} (interpolation)					
P_{inj} (injection)		None	None	None	
P_{ex} (triangular)					

Table 1: The stencils used for the prolongation operators depending on which node is prolongated. The four types of central nodes depend on their neighboring coarse nodes, and are labeled "coarse", "vertical", "horizontal", and "diagonal". White circles are coarse nodes, black dots are fine nodes. A line corresponds to a relation with the central node in the prolongation stencil. Also, we give the reference element for each prolongation type: finite difference for P^l_{l+1} and P_{inj}, $\mathcal{P}_2$ finite element for P_{ex}.

4.2 Smoothing procedure

Pointwise smoothing leads to slow convergence of the multigrid scheme when polar or curvilinear coordinate formulations are considered with standard coarsening [TOS01, Chap. 5.1.1]. [Bar88] proposed the use of circle- and radial-line smoothing operations each using two alternating colors (black and white) for every line on the full unit disk or annulus. On an annulus, for compact stencils (i.e., stencils of a maximum length of one in each direction), all lines of one color of a particular smoother (i.e., circle or radial) can be considered in parallel. On full disks, special care has to be taken for radial smoothing as its colors get connected by the origin; see also Figure 4 for a motivation.

Due to the anisotropy introduced by the coordinate transformation, circle and radial smoothing behave differently on different parts of the computational domain. This means that smoothing each node with both smoothers, i.e. first all nodes with the first smoother and then all nodes with the second one, may be inefficient in terms of computational cost. Based on the results of [Bar88], in [KKR22], it was numerically shown that one smoothing step per grid node (either radial or circle) can be sufficient to obtain fast convergence. Therefore, we partition the domain into two subdomains where either circle or radial smoothing is applied. The criterion to change from circle (starting with the innermost circle line) to radial smoothing is to find a minimum i such that it exists a j with

$$\frac{k_j}{h_i} r_i > 1; \tag{26}$$

see Figure 3 for an illustration.

We define n_μ as the index of the outermost radius of the interior subdomain. In Figures 4 and 3, we have $n_\mu = 4$ and the corresponding circle would be given by the nodes with global numbers 25 to 32. Then the number of nodes in a black (or white) radial line in the exterior subdomain is $n_r - n_\mu = 4$, e.g., nodes $\{36, 44, 52, 60\}$ in Figure 4.

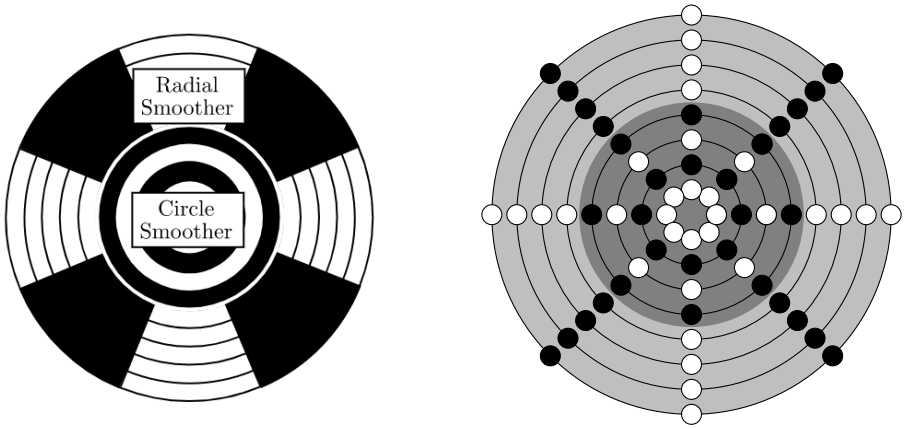


Fig. 3: **Left**: Schematic subdivision of a circular domain into an interior subdomain where circle-line smoothing is used and an exterior subdomain where radial smoothing is used. **Right**: Circular mesh with 64 nodes. Circular smoothing applied on the four interior circles (dark gray background color), radial smoothing applied on the four exterior circles (light gray background color). Nodes colored according to smoother color.

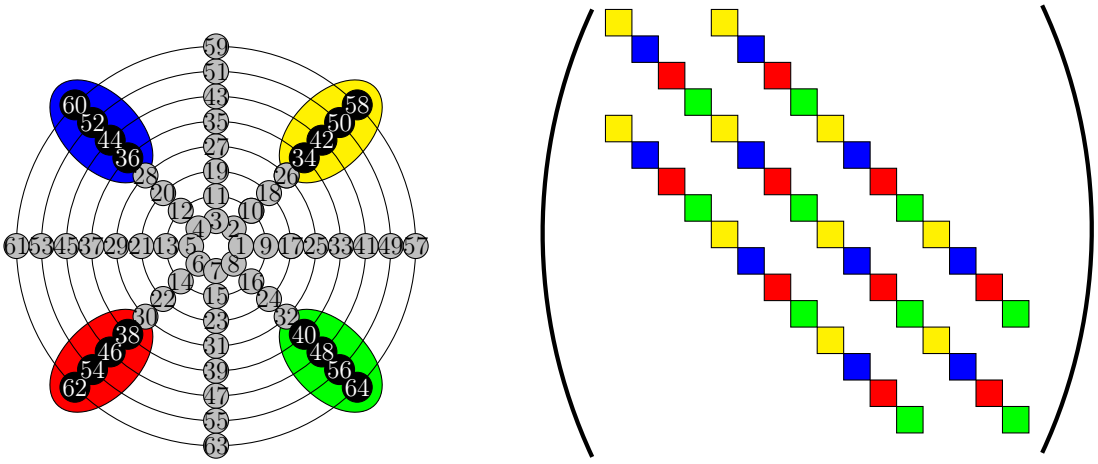


Fig. 4: **Left**: Circular mesh with 64 nodes as in Figure 3 (right). Without loss of generality, we assume a circle-wise global numbering. Focus on black colored nodes of the radial smoother. Different grid lines of the smoother given by different background colors. **Right**: Nonzero pattern of the submatrix $A_{R_B R_B}$ when using a compact five, seven, or nine point stencil, with colors referring to the particular grid lines as depicted on the left figure. Different numberings of the nodes lead to different nonzero patterns.

The line smoother used is a simple block Gauss-Seidel algorithm and these results also hold for more general geometries. The sequential execution of the two partial smoothers (circle and radial), considering the update from the first one in the second one, reduces the iteration numbers. Moreover, a partially parallel implementation can substantially reduce the computation time without increasing the iteration number. For more details, see [KKR22, Sec. 4.1] and Section 6.

We further introduce the double index s_c to refer to a subset of nodes used in partial smoothing operations. We use $s \in \{\mathrm{C}, \mathrm{R}\}$ with C for circle and R for radial as well as $c \in \{\mathrm{B}, \mathrm{W}\}$ with B for black and W for white. For instance, with $s = \mathrm{R}$ and $c = \mathrm{B}$, we have the square submatrix

$$A_{R_B R_B} \tag{27}$$

of A, corresponding to a restriction on the black colored nodes where radial smoothing is applied. In Figure 4, we provide a small example with the R_B nodes highlighted with different background colors, and the corresponding nonzero pattern of the submatrix $A_{R_B R_B}$. Note that $A_{R_B R_B}$ can easily be reordered as a block diagonal matrix when using, e.g., compact five, seven, or nine point stencils. In this case, all lines can be handled in parallel. Similarly to A, the part of the solution vector (resp. right hand side) corresponding to the nodes on the s_c lines is denoted by u_{s_c} (resp. f_{s_c}). Finally, we define

$$\mu_s := m_s/m \quad \text{and} \quad \mu_{s_c} := m_{s_c}/m, \tag{28}$$

the ratios of unknowns touched by the partial smoothing against the total number of unknowns on the current grid. Here, m_s represents the number of unknowns treated by the partial smoothing procedure s (both colors for either circle or radial smoothing steps), while m_{s_c} refers to the subset of unknowns treated by only one color c. Similarly, we can define $\mu_c := m_c/m$ as the number of unknowns of color c (either black or white), and we have $m_B \approx m_W \approx m/2$.

For the consideration of the complexity of the overall algorithm, we break all the ingredients down to the smallest possible unit. Therefore, let us further consider a submatrix of the previously introduced submatrix $A_{s_c s_c}$. For both partial smoothers, a *grid line* refers to one line that is smoothed together. In Figure 4 (left), we have four radial black grid lines, highlighted by the different background colors yellow, blue, red, and green. Let us denote the submatrix corresponding to the i-th grid line

$$A^{(i)}_{s_c s_c}. \tag{29}$$

For instance, we obtain the matrix $A^{(0)}_{R_B R_B}$ by extracting all yellow squares from Figure 4 (right). Similarly, we write $f^{(i)}_{s_c}$ for the corresponding right hand side part. The matrix $A^{(i)}_{s_c s_c^\perp}$ then consists of all remaining elements, which are the rows corresponding to the i-th grid line, and all columns that do not belong to s_c; that is, either they belong to the other color of the same partial smoother or any color of the other partial smoother. One relaxation step for this line is then expressed as

$$u^{(i)}_{s_c} = \left(A^{(i)}_{s_c s_c}\right)^{-1} \left(f^{(i)}_{s_c} - A^{(i)}_{s_c s_c^\perp} u^{(i)}_{s_c^\perp}\right). \tag{30}$$

Table 2 gives the stencils of $A_{s_c s_c}$ and $A_{s_c s_c^\perp}$ for each type of partial smoother.

Finally, a complete relaxation step in GMGPolar applies both partial smoothers $s \in \{C, R\}$ with both colors, black (B) and white (W). Different colors are treated sequentially while all lines of one color are smoothed in parallel. That means, e.g., that the scheme of the second radial color uses the update from the Gauss-Seidel scheme of the first radial color.

4.3 Multigrid iterations and implicit extrapolation

Using the previously introduced hierarchy of systems and the tailored smoother on each level, GMGPolar follows a traditional V-cycle to solve the discretized system. Algo. 1 gives the whole V-cycle process, where $\mathcal{S}_\nu(u, A_l, f_l)$ returns the result of ν relaxation steps applied onto u on level l.

Algorithm 1 $\mathbf{MG}(u_l,\ l)$: Multigrid V-cycle with ν_1 pre-smoothing and ν_2 post-smoothing steps on level l (L levels with $l = 1$ the finest)

if coarsest level **then**
  return $A_L^{-1} u_L$ `(exact solve)`
else
  $u_l \leftarrow \mathcal{S}_{\nu_1}(u_l, A_l, f_l)$ `(pre-smoothing)`
  $r_{l+1} \leftarrow P_{l+1}^{l\ T}(f_l - A_l u_l)$ `(restriction of the residual)`
  $e_{l+1} \leftarrow \mathbf{MG}(r_{l+1}, l+1)$ `(coarse grid correction)`
  $e_l \leftarrow P_{l+1}^{l} e_{l+1}$ `(interpolation of the error)`
  $u_l \leftarrow u_l + e_l$ `(update of the solution)`
  return $\mathcal{S}_{\nu_2}(u_l, A_l, f_l)$ `(post-smoothing)`
end if

In [KKR21], the previously introduced variant of GMGPolar showed an approximation error of degree 2 (compared to a manufactured solution of the Poisson-like problem (1) with $\beta = 0$) in a modest number of iteration steps. In order to increase the approximation order of the algorithm from order 2 to order 3 or 4, the authors in [KKR22] presented a mechanism called implicit extrapolation [Rüd91, JR96] which can naturally be implemented in the multigrid scheme. The implicitly extrapolated multigrid is based on the τ-extrapolation technique [Ber97]. For more details, see the motivation in [Sch21, Sec. 4.5] and the references therein. This approach allows to increase the approximation order without the explicit construction of a higher order discretization, and thus, keeps the modest computational cost of the original multigrid method GMGPolar.

The implicit extrapolation uses additional information from the next coarser grid to estimate the local error, and to eliminate certain dominating error terms on the finest level. Due to the suitable combination of lower order systems when computing the new residual, we can implicitly achieve a higher order solution. Note that with implicit extrapolation, we only relax the fine nodes on the finest level, ensuring that the multigrid iteration converges to a unique higher order solution. For more details, see [JR96].

For this paper to be self-contained, we briefly recall how the components of the multigrid cycle are affected by this method. The extrapolation technique is only applied between the finest two grids. Without loss of generality, let us assume that the unknowns are ordered such that the coarse nodes are ordered first and the remaining fine (or fine-only) nodes second.

As the extrapolation only affects the first two levels, let us drop the index l of $A_l u_l = f_l$ or explicitly set it to $l = 1$ for (31)-(33). The system $Au = f$ then writes

$$\begin{pmatrix} A_{f^\perp f^\perp} & A_{f^\perp f} \\ A_{f f^\perp} & A_{ff} \end{pmatrix} \begin{pmatrix} u_{f^\perp} \\ u_f \end{pmatrix} = \begin{pmatrix} f_{f^\perp} \\ f_f \end{pmatrix}, \tag{31}$$

where the indices $\cdot_f$ and $\cdot_{f^\perp}$, respectively correspond to fine nodes which do not belong to the underlying coarse mesh, and the coarse nodes. The multigrid cycle with implicit extrapolation then differs in two aspects.

First, and contrary to the τ-extrapolation [Ber97], the smoothing is only performed on the unknowns u_f on the fine grid excluding the coarse unknowns $u_{f^\perp}$; for more details see [JR96]. Thus, a relaxation step follows Equation (30) but, for each line, the matrix A_{s_c} is restricted to the fine unknowns u_f, and $A_{s_c}^\perp$ also affects the coarse unknowns $u_{f^\perp}$ of the lines s_c, see Table 2. We denote this smoothing procedure by

$$\mathcal{S}_{\nu_1}\big|_f(u, A, f) := \begin{pmatrix} u_{f^\perp} \\ \mathcal{S}_{\nu_1}(u_f, A_{ff}, f_f - A_{ff^\perp} u_{f^\perp}). \end{pmatrix} \tag{32}$$

Secondly, we use a different intergrid transfer operator between the two finest levels. The restricted residual on the first level ($l = 1$) of the implicitly extrapolated scheme is given by

$$r_{ex} := \frac{4}{3} P_{ex}^T (f_1 - A_1 u_1) - \frac{1}{3} (f_2 - A_2 P_{inj}^T u_1). \tag{33}$$

Here, P_{inj} is the simple injection operator such that ${P_{inj}}^T$ extracts the coarse nodes from u_1. The operator P_{ex} is specific to the extrapolation approach. P_{ex} is based on triangle finite element theory and for more details, see [KKR22, (4.8)] as well as [KKR21]. It is the identity operator on the coarse nodes. For the remaining fine nodes, it returns 1/2 of the values of the adjacent coarse nodes in the triangle finite element. Table 1 gives the stencil corresponding to P_{inj} and P_{ex}.

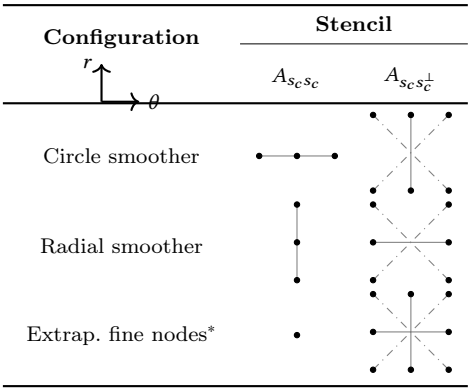


Table 2: The stencils used for the smoother matrices $A_{s_c s_c}$ and $A_{s_c s_c^\perp}$. A black dot corresponds to a relation with the central node in the stencil. *: "Extrap. fine nodes" refers to the stencil used when smoothing fine nodes (of the same color as the coarse nodes) with the implicitly extrapolated scheme, identically for both circle and radial smoothing.

The complete implicitly extrapolated multigrid cycle is given in Algo. 2. In the context of finite elements, it can be shown that the discretization, with linear elements and extrapolation, is equivalent to a quadratic discretization when nonstandard integration rules are applied. For more details, see [Rüd91, JR98, KKR21]. In practice, approximation orders between 3 and 4 are observed for the finite difference scheme introduced in [KKR21, KKR22], where the authors use $\beta = 0$ in Equation (1). In Section 7, we demonstrate similar approximation orders using a general value for the coefficient β, and the discretization from Section 3.

Algorithm 2 iexMG(u_1): Implicit extrapolation applied on the finest level $l = 1$ with ν_1 pre-smoothing and ν_2 post-smoothing steps

1: $u_1 \leftarrow \mathcal{S}_{\nu_1}\big|_f (u_1, A_1, f_1)$ `(pre-smoothing fine nodes)`
2: $r_{ex} \leftarrow \frac{4}{3} P_{ex}^T (f_1 - A_1 u_1) - \frac{1}{3} \left(f_2 - A_2 P_{inj}^T u_1 \right)$ `(extrapolated residual)`
3: $e_2 \leftarrow \mathbf{MG}(r_{ex}, 2)$ `(coarse grid correction)`
4: $e_1 \leftarrow P_{ex} e_2$ `(interpolation of the error)`
5: $u_1 \leftarrow u_1 + e_1$ `(update of the solution)`
6: return $\mathcal{S}_{\nu_2}\big|_f (u_1, A_1, f_1)$ `(post-smoothing fine nodes)`

5 Computation and complexity

Numerical experiments are an essential tool to allow considerations which are difficult or impossible to realize by physical experiments. However, when it comes to the application of numerical algorithms on supercomputers, their cost in terms of computations and memory can also become important. In order to keep the cost reasonable and to make best use of the resources, these algorithms should be designed with care. This means that potential bottlenecks should be investigated and possibly removed or reduced. Here, we solve the related problem (1) with a target size between millions and a billion of dofs, mainly with a shared memory approach. However, many hundreds or thousands instances of these problems have to be solved in each time step of a gyrokinetic code such as Gysela [GAB+16].

Multigrid methods can be asymptotically optimal, in the sense that the complexity to solve a linear system with sufficient accuracy grows only linearly with the number of unknowns m. This has been shown for the full multigrid method [BL11]. Let us recall that GMGPolar follows a matrix-free implementation in order to decrease the memory-footprint, thus a problem of a size up to a billion can be solved in a single computing node, at the expense of a higher computational cost. In this section, we provide the theoretical complexity of GMGPolar in terms of computation and memory consumption. In particular, we show that GMGPolar inherits an optimal complexity. The computation of such complexity is rarely done for complete solvers. This section, completed by the appendix Appendix 8.4.1, can serve as a general approach for other frameworks.

Remark 3 The computational complexity of our method is expressed here in terms of *floating point operations* (flops). Although we are aware that different operations do not correspond to the same number of cycles on an actual machine, e.g. the addition or the division, we consider here that they are all equivalent. This number of flops still gives a fair idea of how expensive an algorithm is.

5.1 General considerations

Let us consider $m = n_r \times n_\theta$, the number of rows and columns of the matrix, as well as the number of nodes (i.e., degrees of freedom) in the grid, on the finest level. Our goal here is to obtain an estimation of the asymptotic complexity with respect to m, in terms of both memory and computation. We denote by $\mathfrak{F}(\mathbb{X})$ the computational cost of the operation $\mathbb{X}$, and by $\mathfrak{M}(\mathbb{V})$ the memory cost of the structure (or the function) $\mathbb{V}$. Since we are interested in the asymptotic complexity of GMGPolar, we simplify the calculus by neglecting the boundaries (interior radius R_{min}, exterior radius R_{max}, as well as the boundary between radial and circle-line smoothers). We also neglect the manipulation of integers, to only focus on floating point numbers and their operations of the order of $O(m^i)$, $i \in \mathbb{N}^*$.

In order to compute the complexity on the whole grid, we often consider the complexity per node $\mathfrak{F}_m(\mathbb{X})$ and $\mathfrak{M}_m(\mathbb{V})$. If the complexity applies to all nodes likewise (except the effect of boundaries), we have for the overall complexity

$$\mathfrak{F}(\mathbb{X}) = O\left((\mathfrak{F}_m(\mathbb{X})m\right) \quad \text{and} \quad \mathfrak{M}(\mathbb{V}) = O\left(\mathfrak{M}_m(\mathbb{V})m\right). \tag{34}$$

Let us start with an example: As input, we need to have a *right hand side* (RHS) vector of size m defined on the finest level. We consider it as an input vector of size m. Then, $\mathfrak{F}(RHS) = 0$, and the corresponding memory consumption is $\mathfrak{M}(\text{RHS}) = m$. Also, we need to store the mesh information, i.e. the grid, on all levels. As we use a tensor-format mesh, the grid is represented by two arrays of respective size n_r and n_θ containing the divisions in the r- and θ-directions.

For a non-degenerated grid, we assume $n_r \approx n_\theta \approx \sqrt{m}$. Because the size of the r- and θ-arrays are of order $\sqrt{m}$, we choose to neglect them to estimate the asymptotic complexity and memory consumption. In order to decrease the computations, we also consider that we have precomputed the values of $\cos(\theta)$ and $\sin(\theta)$ in the setup phase (only dependent on θ) as well as the values of the coefficients α and β from Equation (1) (only dependent on r). The corresponding number of flops and memory consumption is of the order $\sqrt{m}$, and thus neglected.

Let us now consider a matrix M, which entries M_{ij} can be written as the sum of several factors, i.e. $M_{ij} = \sum_k M_{ij}^{(k)}$. The computational cost to build M, denoted by $\mathfrak{F}(build\ M)$, then includes the cost to build each factor $M_{ij}^{(k)}$, as well as an addition to add this factors into M_{ij}. Furthermore, considering a vector u and the vector $v = Mu$, we have for each entry $v_i = \sum_j M_{ij}u_j = \sum_{j,k}\sum_k (M_{ij}^{(k)})u_j$. The computational cost to build Mu includes one addition and one multiplication for each factor $M_{ij}^{(k)}$ composing the matrix. In a matrix-free approach, each entry of the matrix must be recomputed on-the-fly at each application, since the matrix is not kept in memory. As a result, the computational cost for the matrix-free application of M can be decomposed in the cost to construct the matrix (including one addition per factor), plus a cost to apply the matrix which corresponds only to the multiplication by the entries of u (the addition is already accounted for). We write that

$$\mathfrak{F}(matrix\text{-}free\ M) = \mathfrak{F}(build\ M) + \mathfrak{F}(apply\ M), \tag{35}$$

where $\mathfrak{F}(apply\ M)$ is simply the number of multiplications by u_i, i.e. the total number of different factors $M_{ij}^{(k)}$ composing the matrix. This result is true for the matrix A, as well as the prolongation and smoothing operators.

We now study the complexity of the three main components performed in the multigrid cycle detailed in Algo. 1 and 2: the computation of the residual (including the restriction), the smoothing procedure, and the coarsest grid solve. We first consider that the implicit extrapolation is turned off, and will detail the differences in Section 5.5. In order to lighten this very technical section, we only provide the main steps in order to compute the final complexity here. The whole computation is added as an appendix. The final complexity estimates are given in Table 3.

5.2 Computation of the residual

The computation of the residual mainly involves the matrix-free application of the matrix A and the restriction P^T.

5.2.1 Matrix-free application of A

As introduced in Section 3, the matrix A is based on a finite difference scheme with a nine point stencil in GMGPolar. When applying A, nine entries must be computed for each node in the grid. These entries imply the computation of the three functions a^{rr}, $a^{\theta\theta}$, and $a^{r\theta}$ on the current node and its neighboring nodes, from Equation (19), which we generically denote by a^{xx} .

In order to apply A, we distinguish two approaches, which are illustrated in Figure 5 for the computation of a stencil. In a naive approach, here called $A - take$, the algorithm loops over all the grid nodes and computes the entries of the stencil directly, including the a^{xx} values for the neighboring nodes. Overall, the values for a^{rr} and $a^{\theta\theta}$ must be computed three times for each

Table 3: Cost and number of applications for the functions used in the multigrid cycle. For each group of functions (surrounded by horizontal lines), we provide a summary statistics in bold face. If memory gets reused, the summarized memory is less than the sum of the previous rows. For computations, the summary statistics is the sum of the previous rows. ν_1 and ν_2 are the number of pre- and post-smoothing sweeps. We denote ξ the number of iterations for the convergence of the multigrid cycle.

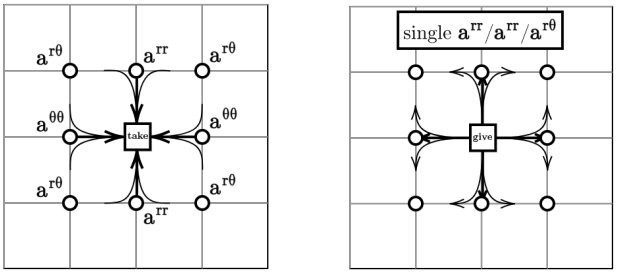
Function	Memory	Computation	Applications	Levels
right hand side	m	input	–	$l = 1$
matrix-free Au	m	$\mathcal{O}([\mathfrak{F}_m(DF) + 99]m)$	ξ	$l = 1, \cdots, L-1$
matrix-free $P^T u$	m	$\mathcal{O}([7.25]m)$	ξ	
residual	$3m$	$\mathcal{O}([\mathfrak{F}_m(DF) + 107.25]m)$	ξ	$l = 1, \cdots, L-1$
build A_L	$\mathcal{O}(9m_L)$	$\mathcal{O}(\mathfrak{F}_m(DF) + 74)$	1	$l = L$
factorization A_L	$\mathcal{O}\left(m_L^2/2\right)$	$\mathcal{O}\left(m_L^3/3\right)$	1	
coarse setup	$\mathcal{O}\left(m_L^2/2\right)$	$\mathcal{O}\left(m_L^3/3 + \mathfrak{F}_m(DF) + 74\right)$	1	$l = L$
compute $A_L^{-1} z$	m_L	$\mathcal{O}\left(2m_L^2 - m_L\right)$	ξ	$l = L$
coarse apply	m_L	$\mathcal{O}\left(2m_L^2 - m_L\right)$	ξ	$l = L$
build $A_{s_c s_c}$	$3m$	$\mathcal{O}([2\mathfrak{F}_m(DF) + 67]m)$	1	$l = 1, \cdots, L-1$
factorization $A_{s_c s_c}$	$\mathcal{O}([3 + 2\mu_C]m)$	$\mathcal{O}([3 + 7\mu_C]m)$	1	
smoother setup	$\mathcal{O}([3 + 2\mu_C]m)$	$\mathcal{O}([2\mathfrak{F}_m(DF) + 70 + 7\mu_C]m)$	1	$l = 1, \cdots, L-1$
compute $A_{s_c s_c}^{-1} z$	m	$\mathcal{O}([5 + 4\mu_C]m)$	$(\nu_1 + \nu_2)\xi$	$l = 1, \cdots, L-1$
matrix-free $A_{s_c s_c^\perp} u_{s_c^\perp}$	m	$\mathcal{O}([2\mathfrak{F}_m(DF) + 74]m)$	$(\nu_1 + \nu_2)\xi$	
smoother apply	$2m$	$\mathcal{O}([2\mathfrak{F}_m(DF) + 80 + 4\mu_C]m)$	$(\nu_1 + \nu_2)\xi$	$l = 1, \cdots, L-1$

node, i.e., for the stencil of the current node and the two neighbors respectively from the same r and θ. The values for the $a^{r\theta}$ must be computed four times. In another approach, called $A-give$, we loop over all nodes to compute their local a^{xx} values only once, and update the entries in the matrix corresponding to the neighboring nodes. Thus, we directly divide by around three the number of computations for the functions a^{xx} . The functions a^{xx} are composed of similar elements, and we further decrease the computational cost by computing them simultaneously for each node.

Remark 4 Based on the stencil definition from Equations (19)–(21), the $a^{r\theta}$ functions used for the diagonal updates originate from the neighbors with the same r or θ as the current node, thus the curved arrows in Figure 5. In other words, we never need $a^{r\theta}$ function evaluations at the diagonal neighbors' positions. In the rest of the paper, we simplify the representation by drawing diagonal arrows for these relations.

Remark 5 In a finite element context, the A-give approach would correspond to looping over the grid nodes and adding in the matrix the contributions of surrounding elements. The A-take approach amounts to looping over the elements to update at once all the entries of the stiffness matrix corresponding to its vertices.

Remember that the values of $\cos(\theta)$, $\sin(\theta)$, α, and β are precomputed and their cost is asymptotically negligible as they only depend on either θ or r. The core element of the functions a^{xx} is the computation of the Jacobian DF of the transformation F, whose fixed cost depends



(a) *A-take* (b) *A-give*

Fig. 5: Schematic representation of the two approaches to apply the matrix A: *take* or *give* the evaluations of the functions a^{xx} evaluations from or to the neighboring nodes, respectively.

on the chosen geometry. The computation of the three functions a^{xx} can then be obtained for $\mathfrak{F}_m(a^{xx}) = 19 + \mathfrak{F}_m(DF)$ flops per node, see Appendix 8.2. If a subset of the a^{xx} is not needed, its specific cost is subtracted from the total sum (4 flops for a^{rr} or $a^{\theta\theta}$, and 5 flops for $a^{r\theta}$) but there remains an irreducible common cost of 6 flops corresponding to DF.) Additionally, by computing at once the a^{xx} of a whole line, with a fixed radius, it is possible to benefit from the vectorization of modern computing architectures in order to increase the computing intensity.

Adding the small cost from computing the scaling coefficients in Equations (19)–(21) (see Appendix 8.3) gives the complexity of building the matrix A, i.e.,

$$\mathfrak{F}_m(build\ A) = \mathfrak{F}_m(DF) + 74 \text{ flops}. \tag{36}$$

As explained in Section 5.1, in order to apply A to a vector in a matrix-free fashion, each factor composing the entries of A is multiplied with the corresponding vector value resulting in $\mathfrak{F}_m(apply\ A) = 25$ flops. Finally, with (35) and its surrounding explanation, we get

$$\mathfrak{F}_m(matrix\text{-}free\ Au) = \mathfrak{F}_m(build\ A) + \mathfrak{F}_m(apply\ A) = \mathfrak{F}_m(DF) + 99 \text{ flops}. \tag{37}$$

5.2.2 Prolongation operator

The prolongation operator used in GMGPolar is a bilinear interpolation handling non-uniform grids. The stencil of this operator differs depending on the type of the prolongated node, i.e. depending on the neighboring coarse nodes. As defined in Table 1, there are four types of nodes which updates are defined as

$$u_l(r_p, \theta_q) = \begin{cases} u_{l+1}(r_p, \theta_q) \quad \texttt{(coarse node injection)}, & \text{if } p \text{ odd and } q \text{ odd}, \\ \frac{1}{k_q + k_{q-1}} \left[k_q u_{l+1}(r_p, \theta_{q-1}) + k_{q-1} u_{l+1}(r_p, \theta_{q+1}) \right], & \text{if } p \text{ odd and } q \text{ even}, \\ \frac{1}{h_p + h_{p-1}} \left[h_p u_{l+1}(r_{p-1}, \theta_q) + h_{p-1} u_{l+1}(r_{p+1}, \theta_q) \right], & \text{if } p \text{ even and } q \text{ odd}, \\ \frac{1}{(h_p + h_{p-1})(k_q + k_{q-1})} \left[h_p k_q u_{l+1}(r_{p-1}, \theta_{q-1}) \right. & \\ + h_p k_{q-1} u_{l+1}(r_{p-1}, \theta_{q+1}) + h_{p-1} k_q u_{l+1}(r_{p+1}, \theta_{q-1}) & \\ \left. + h_{p-1} k_{q-1} u_{l+1}(r_{p+1}, \theta_{q+1}) \right], & \text{if } p \text{ even and } q \text{ even}. \end{cases} \tag{38}$$

By simple calculus, for a fine node (r_p, θ_q), the number of flops in order to build and apply P are respectively

$$\mathfrak{F}_m(build\ P) = \begin{cases} 0, \\ 4, \\ 4, \\ 12, \end{cases} \quad \text{and} \quad \mathfrak{F}_m(apply\ P) = \begin{cases} 1, & \text{if } p \text{ odd and } q \text{ odd,} \\ 2, & \text{if } p \text{ odd and } q \text{ even,} \\ 2, & \text{if } p \text{ even and } q \text{ odd,} \\ 4, & \text{if } p \text{ even and } q \text{ even.} \end{cases} \tag{39}$$

Since each type of update concerns around $m/4$ nodes, we finally obtain that

$$\begin{aligned} \mathfrak{F}_m(build\ P) = 5 \quad \text{and} \quad \mathfrak{F}_m(apply\ P) = 2.25, \\ \mathfrak{F}_m(matrix\text{-}free\ Pu) = \mathfrak{F}_m(build\ P) + \mathfrak{F}_m(apply\ P) = 7.25. \end{aligned} \tag{40}$$

As for the restriction operator, i.e. $R = P^T$, we have the same complexity. The only difference is that we loop over the fine nodes of the grid to apply the prolongation, and we loop over the coarse nodes to apply the restriction. This way, the update of a specific node is completely independent from the others, which gives a natural and efficient parallelization.

5.2.3 Full residual expression

Finally, the restricted residual is expressed as $P^T(f - Au)$. In terms of memory consumption, only the right-hand side, the solution and the residual vectors are needed (the restricted residual becomes the right-hand side on the next coarser level). We then have

$$\mathfrak{M}(residual) = 3m. \tag{41}$$

And, by taking into account the difference between vectors f and Au in the residual (1 flop per node), we obtain the final complexity $\mathfrak{F}_m(residual) = \mathfrak{F}_m(matrix\text{-}free\ Au)+1+\mathfrak{F}_m(matrix\text{-}free\ Pu)$ for the restricted residual $P^T(f - Au)$, as given in Table 3.

5.3 Smoothing

We now turn to the application of the smoother, i.e., determining how expensive is the application of Equation (30) to each line. The matrices $A_{s_c s_c}$ and $A_{s_c s_c^\perp}$ are submatrices of A, up to some reordering of the rows and columns. We construct and/or apply them following an A-give approach. Thus, we obtain similar computational complexities for their construction and application as for A, see Appendix 8.4.1 for the full details.

Based on the stencils from Table 2, not all of the functions a^{xx} need to be computed for the operators $A_{s_c s_c}$ and $A_{s_c s_c^\perp}$. The matrices $A_{s_c s_c}$ are fully assembled in a setup phase in order to apply $A^{-1}_{s_c s_c}$ during the smoothing steps. In this case, only the a^{rr} and $a^{\theta\theta}$ functions need to be computed. As for the application of $A_{s_c s_c^\perp}$, the $a^{r\theta}$ functions are needed for all nodes, however, either the computation of the a^{rr} or the $a^{\theta\theta}$ functions are required for the circle and the radial partial smoothers respectively.

Again, it is possible to group the computation of the functions a^{xx} and use vectorization. Although for the matrix A, whole circle lines were computed at once, here we group circle lines or radial lines respectively for each partial line smoother.

In order to build the matrix $A_{s_c s_c}$ (resp. apply $A_{s_c s_c^\perp}$), we need to loop over the nodes corresponding to the partial smoother s_c but also over their neighbors, i.e. the nodes of the same partial smoother s but with the other color. For example, in an A-give approach, to apply the

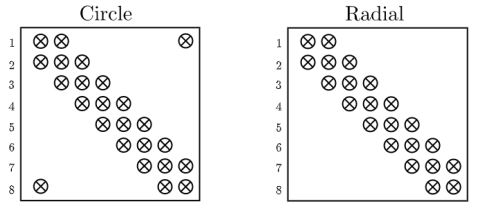
smoother C_B, we also loop over the nodes C_W. Then, after applying all the partial smoothers, we have looped over all the grid nodes twice, implying to compute the Jacobian DF of the mapping twice. Additionally, while the stencil of the $A_{s_c s_c}$ is greatly reduced, a major part of the computations from A is kept because the central update of the stencil is the sum of the left, right, top, and bottom updates. Thus, the computational cost corresponding to the application of $A_{s_c s_c^\perp}$ is similar to the one for the construction of $A_{s_c s_c}$, with only the additional diagonal updates.

In the end, setting aside the computation of the Jacobian DF which must be performed twice per node, the computational cost to build $A_{s_c s_c}$ and to apply $A_{s_c s_c^\perp}$ represent respectively $2/3$ and $3/4$ of the operations required to build A, see Table 3 and Appendix 8.3.

Since the computational cost related to a^{rr} and $a^{r\theta}$ is the same, the cost of the circle and radial-line smoothers are identical. In fact, in terms of computational cost, the stencil of the radial partial smoother is the same as a 90° rotation of the stencil for the circle partial smoother with.

Once $A_{s_c s_c}$ has been built, the relaxation step is applied at each iteration, for each type of partial smoother. Considering each line of the smoothers separately as in Equation (30), the structure of the corresponding matrix $A^{(i)}_{s_c s_c}$ is specific to the partial smoother used:

- for the circle-line smoother, $A^{(i)}_{s_c s_c}$ is a tridiagonal matrix with periodicity condition, see Figure 6a,
- for the radial-line smoother, $A^{(i)}_{s_c s_c}$ is a tridiagonal matrix, see Figure 6b. In the actual GMGPolar, the Dirichlet boundary condition would also appear in the form of a 1 as first entry in the diagonal but we neglect this aspect for the computation of the asymptotic complexity.



(a) circle-line smoother (b) radial-line smoother

Fig. 6: Sparsity pattern of the matrix $A_{s_c s_c}$ for a line of eight nodes.

Using a modified Thomas' algorithm, we then compute and store an LU decomposition of the $A_{s_c s_c}$ matrices. For one line of n unknowns, the computational cost of the factorization is then respectively $\mathcal{O}(10n)$ and $\mathcal{O}(3n)$ for the circle and the radial-line smoothers. See Appendix 8.4.2 for the algorithms. The total computational cost to factorize the $A_{s_c s_c}$ matrices is obtained by summing on all lines and all partial smoothers. Considering μ_C and μ_R the ratio of unknowns associated with each type of smoother with respect to m, we obtain

$$\mathfrak{F}_m(facto.\ A_{s_c s_c}) = 10\mu_C + 3\mu_R = 3 + 7\mu_C, \tag{42}$$

since $\mu_C + \mu_R = 1$. The latter cost is an abstraction giving the average cost per node to compute the LU factorization. While there is no fill-in associated to the factorization of the matrix for the radial smoother, the last row and column of the matrix will be filled for the LU decomposition of the circle-line smoother. The LU decomposition of the matrices $A_{s_c s_c}$ are stored in a sparse

matrix format, replacing the original $A_{s_c s_c}$ in memory. Accounting for the fill-in, the associated memory consumption is

$$\mathfrak{M}_m(facto.\ A_{s_c s_c}) = 3 + 2\mu_C. \tag{43}$$

At each relaxation, simple forward and backward substitutions are then applied to solve Equation (30), based on the computed LU factorizations, for a cost of $\mathcal{O}(9n)$ and $\mathcal{O}(5n)$ respectively for the circle and radial smoothers. The result is

$$\mathfrak{F}_m(compute\ A_{s_c s_c}^{-1} z) = 9\mu_C + 5\mu_R = 5 + 4\mu_C, \tag{44}$$

see Appendix 8.4.2 for the algorithms.

Thus, we obtain the final complexities of the smoothing steps: $\mathfrak{F}_m(smoother\ setup)$ (computed once), $\mathfrak{F}_m(smoother\ apply)$ (at each iteration). As for the corresponding memory consumption, we have $\mathfrak{M}_m(smoother\ setup) = \mathfrak{M}_m(facto.\ A_{s_c s_c})$, then $\mathfrak{M}_m(smoother\ apply) = 2$ corresponding to the two vectors $A_{s_c s_c}^{-1} z$, $A_{s_c s_c^\perp} u_{s_c^\perp}$. Overall, the smoothing procedure costs $\mathfrak{M}_m(smoother) = \mathfrak{M}_m(smoother\ apply) + \mathfrak{M}_m(smoother\ setup)$, as shown in Table 3.

5.4 Solving on the coarsest grid

On the coarsest grid of the multigrid hierarchy, GMGPolar uses a direct solver at each iteration to compute the coarse grid correction e_L through the solution of the equation

$$A_L e_L = r_L, \tag{45}$$

where A_L is the coarsest grid operator of size m_L, and r_L is the restricted residual transferred from the finer grid level $L-1$. For this purpose, during the solver setup phase, the matrix A_L is first constructed with the computational cost $\mathfrak{F}(build\ A_L)$ as detailed in Equation (36). The corresponding memory consumption is $\mathfrak{M}_{m_L}(build\ A_L) = 9$ due to the nine point stencil. Since the operator A_L is *symmetric*, we can then compute its factorization $A_L = \Gamma\Sigma\Gamma^T$, where Γ is lower triangular and Σ is diagonal, for the known cost

$$\mathfrak{F}(facto.\ A_L) = \mathcal{O}(m_L^3/3). \tag{46}$$

This factorization is also computed only once during setup and the Γ factor is stored in a sparse matrix format. Due to the structure of the matrix A, resulting from the use of the nine point finite differences stencil, the factorization fills-in, and the storage of the Γ and Σ factors is $\mathfrak{M}(facto.\ A_L) = \mathcal{O}(m_L^2/2)$. These factors replace A_L in memory. At each iteration, simple forward substitution (solve $\Gamma u = r_L$) and backward substitution (solve $\Gamma^T e_L = \Sigma^{-1} u$) are then applied to solve Equation (45) for $\mathfrak{F}({A_L}^{-1}) = \mathcal{O}(2m_L^2 - m_L)$ flops, which can be neglected compared to the $\Gamma\Sigma\Gamma^T$ factorization above.

Although the computational cost associated with the coarsest grid correction has a non-linear complexity, this is contrasted by the relatively low dimension of the coarsest grid, i.e., $m_L \approx m/4^L$. In order for the cost (46) to become lower than $\mathcal{O}(m)$, we then need the number of grid levels L to respect

$$\frac{m_L^3}{3} < m \quad \Longleftrightarrow \quad L > \frac{\ln\left(\frac{m^2}{3}\right)}{3\ln(4)}. \tag{47}$$

For example, considering a finest system of size $m = 10^6$, the required number of levels is then $L > 6$. There is a trade-off to find between a mild cost to solve on the coarsest grid, and a faster convergence.

5.5 Implicit extrapolation

The implicit extrapolation technique allows to increase the order of approximation of the solution computed in GMGPolar without the explicit use of a higher order discretization. We now detail all changes in complexity involved in the use of the implicit extrapolation, and show in particular that the complexity stays mostly unchanged. We denote by $\mathfrak{F}_m^{ex}(\mathbb{X})$ and $\mathfrak{M}_m^{ex}(\mathbb{V})$ the computational cost of operation $\mathbb{X}$ and memory requirement of data structure $\mathbb{V}$ when using the extrapolation.

Note again that the extrapolation only acts between the two finest levels and that the smoother on the finest level is only applied on the fine nodes. Let's remind the reader that the coarse grids are defined using standard coarsening as detailed in Section 4.1. Thus, starting from the first grid node, the coarse nodes are chosen with a distance of two from each other. Assuming without loss of generality that the first circle line and the first radial line are white, half of their points are coarse nodes, i.e. kept on the next coarser grid level. As defined in (32), these coarse nodes are kept out of the white smoothers. Written in terms of a linear equation, instead of solving (30), we have

$$\begin{pmatrix} I^{(i)}_{s_{W,f^\perp}} & 0 \\ 0 & A^{(i)}_{s_{W,f}s_{W,f}} \end{pmatrix} \begin{pmatrix} u^{(i)}_{s_{W,f^\perp}} \\ u^{(i)}_{s_{W,f}} \end{pmatrix} = \begin{pmatrix} u^{(i)}_{s_{W,f^\perp}} \\ f^{(i)}_{s_{W,f}} - A^{(i)}_{s_{W,f}s^\perp_W} u^{(i)}_{s^\perp_W} - A^{(i)}_{s_{W,f}s_{W,f^\perp}} u^{(i)}_{s_{W,f^\perp}} \end{pmatrix} \tag{48}$$

for each line i of a white-colored smoother. Here, $I_{s_{W,f^\perp}}$ is the identity matrix acting on the coarse nodes and likewise we have denoted subvectors and matrices of A and u. Consequently, we only solve the second line of the previous equation set. In order to simplify the notation, we will in the following generically use $A^{(i)}_{s_W s_W}$ for the nontrivial smoothing matrix $A^{(i)}_{s_{W,f}s_{W,f}}$.

- Using implicit extrapolation, the $A^{(i)}_{s_W s_W}$ matrices are diagonal. In fact, the $A^{(i)}_{s_W s_W}$ matrices correspond to the submatrix of A restricted to the white fine nodes of the line i as also shown in the previous equation. These fine nodes are independent from each other when using the nine-point stencil, i.e. a stencil of length one. Consequently, only the central point of the stencil is needed for each node of this type, i.e. the diagonal entry in the matrix, see Table 2. Thus, the smoother matrices change from tridiagonal to diagonal. Then, setting aside the cost to compute the Jacobian DF, the resulting computational cost represents around 2/3 of the cost without extrapolation. The matrices $(A^{(i)}_{s_W s_W})^{-1}$ are then trivial to apply, thus $\mathfrak{F}_m^{ex}(A^{-1}_{s_W s_W}) = \mu_W$. We only need to keep the diagonals in memory, i.e. $\mathfrak{M}_m(A_{s_W s_W}) = \mu_W$.
- For the coarse white nodes corresponding to each partial smoother, the $A^{(i)}_{s_W s^\perp_W}$ matrices now apply the nine point stencil except for the central update. In fact, $A^{(i)}_{s_c s^\perp_c}$ contains around $\mu_C/2$ additional columns corresponding to the coarse white nodes. The result is an increased cost per line, due to the additional nodes where *all* functions a^{xx} must be computed, and an increased number of corresponding additional updates. However, since the partial smoother is only applied on the fine nodes, $A^{(i)}_{s_W s^\perp_W}$ contains around $\mu_C/2$ less rows when using extrapolation. As a result, $\mathfrak{F}_m^{ex}(\textit{matrix-free}\ A_{s_c s^\perp_c})$ stays unchanged compared to the case without extrapolation, as detailed in Appendix 8.4.1.

As for the black points, the extrapolation does not change the partial smoothers. Let's introduce $\Delta_m^{\mathfrak{F}}(\mathbb{X}) = \mathfrak{F}_m^{ex}(\mathbb{X}) - \mathfrak{F}_m(\mathbb{X})$, and $\Delta_m^{\mathfrak{M}}(\mathbb{V}) = \mathfrak{M}_m^{ex}(\mathbb{V}) - \mathfrak{M}_m(\mathbb{V})$, the overheads from using the extrapolation scheme. Since the number of black and white lines only differs by one at the maximum,

we can write that $\mu_{s_W} \approx \mu_{s_B}$, $\mu_W \approx \mu_B \approx 1/2$. Then, as detailed in Appendix 8.4.1, we obtain

$$\begin{aligned} \Delta_m^{\mathfrak{F}}(\textit{smoother setup}) &= -9.75 - 2.5\mu_C < -9.75, \\ \Delta_m^{\mathfrak{F}}(\textit{smoother apply}) &= -2.5 - 2\mu_C < -2.5, \\ \Delta_m^{\mathfrak{M}}(\textit{smoother}) &= -1.5 - \mu_C < -1.5, \end{aligned} \tag{49}$$

which means that the smoothing procedure is less expensive with the implicit extrapolation.

Furthermore, the implicit extrapolation introduces two prolongation operators. Considering that $m_{l+1} = m_l/4$, the operator P_{inj} simply corresponds to an injection operator and thus

$$\mathfrak{F}_m^{ex}(\textit{build } P_{inj}) = 0 \quad \text{and} \quad \mathfrak{F}_m^{ex}(\textit{apply } P_{inj}) = \left\{ \begin{array}{l} 1, \ \text{if coarse node,} \\ 0, \ \textit{else} \end{array} \right\} = 1/4. \tag{50}$$

The operator P_{ex} is very similar to the classical interpolation P, except when the neighboring coarse nodes are on the diagonals. In this case (for a quarter of the nodes), we only keep the top-left and bottom-right updates as seen in Table 1. Based on Equation (38), these updates cost 8 flops to construct P_{ex}, and 2 flops to apply. We then obtain

$$\begin{aligned} \mathfrak{F}_m^{ex}(\textit{build } P_{ex}) = 4, \quad \text{and} \quad \mathfrak{F}_m^{ex}(\textit{apply } P_{ex}) = 1.75, \\ \mathfrak{F}_m^{ex}(\textit{matrix-free } P_{ex}u) = \mathfrak{F}_m^{ex}(\textit{build } P_{ex}) + \mathfrak{F}_m^{ex}(\textit{apply } P_{ex}) = 5.75. \end{aligned} \tag{51}$$

Finally, the computation of the residual on the coarsest grid changes to Equation (33), and thus

$$\begin{aligned} \Delta_m^{\mathfrak{F}}(\mathbb{X})(\textit{residual}) = &[\mathfrak{F}_m(\textit{matrix-free } Au) + 1 + \mathfrak{F}_m(\textit{matrix-free } P_{ex}u) \\ &+ 1/4(\mathfrak{F}_m(\textit{matrix-free } Au) + 1) + \mathfrak{F}_m(\textit{matrix-free } P_{inj}u)] \\ &- [\mathfrak{F}_m(\textit{matrix-free } Au) + 1 + \mathfrak{F}_m(\textit{matrix-free } Pu)] \\ \approx\ &1/4(\textit{matrix-free } Au). \end{aligned} \tag{52}$$

As a summary, the only differences in terms of computational cost from using the extrapolation is mainly a less expensive construction of $A_{s_c s_c}$ once during setup (minus at least 10 flops per node; depending on the size of the circle smoother domain), and an overhead of a quarter of an application of A per iteration (i.e. the application of A from the coarser grid level) to compute the modified residual. This overhead does not change the overall complexity, in particular when compared to what would be obtained from using an explicit higher degree discretization.

5.6 Complete and asymptotic complexities

Now, we summarize the complexity of the whole multigrid solver, separating the setup from the actual multigrid cycle. We consider that we apply ν_1 and ν_2 pre- and post-smoothing steps and that the convergence is obtained in ξ iterations. With and without extrapolation, we get

$$\begin{aligned} \mathfrak{F}(\textit{Setup}) &= \mathfrak{F}_m(\textit{setup smoother}) \sum_{l=1}^{L-1} m_l + \mathfrak{F}(\textit{setup coarsest}), \\ \mathfrak{F}(MG) &= \xi([(\nu_1 + \nu_2)\mathfrak{F}_m(\textit{smoother apply}) + \mathfrak{F}_m(\textit{residual})] \sum_{l=1}^{L-1} m_l + \mathfrak{F}(\textit{coarsest})), \\ \mathfrak{M}(MG) &= [\mathfrak{M}_m(\textit{residual}) + \mathfrak{M}_m(\textit{smoother})] \sum_{l=1}^{L-1} m_l + \mathfrak{M}(\textit{coarsest}). \end{aligned} \tag{53}$$

Due to standard coarsening, we have $m_l \approx m/4^l$. As we are only considering orders, not exact equality, we use the equal sign and obtain a geometric series expression

$$\sum_{l=1}^{L-1} m_l = m \sum_{l=1}^{L-1} \left(\frac{1}{4}\right)^l = \frac{4 - \frac{1}{4^{L-1}}}{3} m \overset{L\to\infty}{\to} \frac{4}{3} m. \tag{54}$$

By taking a large number of grid levels and thus considering that the coarsest grid cost can be neglected, we obtain the linear asymptotic complexities

$$\begin{aligned} \lim_{L\to\infty} \mathfrak{F}(Setup) &= \frac{4}{3} \mathfrak{F}_m(setup\ smoother) \times m, \\ \lim_{L\to\infty} \mathfrak{F}(MG) &= \frac{4\xi}{3} [(\nu_1 + \nu_2) \mathfrak{F}_m(smoother\ apply) + \mathfrak{F}_m(residual)] \times m, \\ \lim_{L\to\infty} \mathfrak{M}(MG) &= \frac{4}{3} [\mathfrak{M}_m(residual) + \mathfrak{M}_m(smoother)] \times m \approx 12m. \end{aligned} \tag{55}$$

The number of iterations was empirically shown to be independent of the problem size in [KKR22], and the complexity of GMGPolar is thus optimal in both computations and memory.

6 Parallel implementation

In the previous section, we have demonstrated optimal linear complexity for GMGPolar, in both memory and computations, using suited implementations. Also, most of the operations can be vectorized as explained in the previous section. The implementation has laid the base for parallelization as, e.g., the prolongation operator updates all of its entries independently and the matrix operator updates all neighboring nodes by avoiding duplicate computations. In order to make use of the shared memory infrastructure of modern computers, we will detail a multihreading approach for GMGPolar. This parallelization is implemented with the *OpenMP-4.5* API.

6.1 Application of the intergrid transfer operators

The prolongation operator is the simplest operator to parallelize in our case. Its application to a vector is performed by updating all entries of the output vector independently. In fact, all entries in the output vector correspond to a fine node, and the values are computed as a weighted sum of the neighboring coarse nodes values; see Equation (38). However, updating all nodes in parallel can end up with each thread having too little computation to perform, we say that the granularity may be too low. We can then improve the parallelization by grouping the fine nodes of each row (with a fixed radius), which also has the advantage of vectorizing the computation by grouping memory accesses.

As for the restriction operator, the idea is the same but the loop is performed over the coarse nodes instead of the fine nodes, thus updating again the entries of the output vector independently.

6.2 Matrix-free application of A

Concerning the application of the matrix A to a vector u, we could implement the same kind of parallelization by updating all entries of the output vector independently, i.e. using the $A-take$ approach with duplicate computations of the functions a^{xx} , as introduced in Section 5.2.1.

Using the $A-give$ approach, we avoid duplicate computations of the functions a^{xx} , by looping over all nodes to update the neighboring nodes in the nine point stencil. Then, each entry of the output vector, corresponding to a single grid node, is updated by all of its 8 neighbors. In terms of parallelization, this introduces concurrent memory accesses between threads, and means that we need some mechanism to prevent incoherence from such conflicts. We solve this issue by using *task-based parallelism* and *dependencies*, respectively introduced in the revisions 3.0 and 4.0 of OpenMP.

The principle of task-based parallelism is to separate parts of a complete code as tasks. These tasks are put in a pool of tasks by the master thread. At runtime, the OpenMP threads can pick and perform tasks as they become available, and so on until all the tasks have been accomplished. Additionally, we can explicitly define dependency relations between tasks in order to avoid conflicts from parallel updates. OpenMP interprets these dependencies by creating a *direct acyclic graph* (DAG), named *dependency graph* where each vertex is a task, and the edges define dependencies between these tasks. The threads can only run tasks which dependencies have been resolved before. We obtain an asynchronous mechanism, which is essential to keep a high level of parallelism and efficiency of our application on modern supercomputers.

As before, to prevent issues of granularity and use vectorization, we consider grouping the computations corresponding to one circle line of the grid, i.e., a line with constant radius. We then define as task $t^{(i)}$ the sequential loop over all nodes in the line i with radius r_i. In this loop, we update the parts of Au corresponding to the neighbors of the nodes in line i, i.e. all nodes in the lines i, $i-1$, and $i+1$. Since the nodes are considered sequentially inside a row, there can be no conflict of update between two nodes of the same row. However, row i is also updated by the rows $i-1$ and $i+1$, and thus, there can be conflicts from concurrent updates between these three rows. We say that the tasks $t^{(i)}$, $t^{(i-1)}$, and $t^{(i+1)}$ are dependent from each other. Consequently, we have that $t^{(i)}$ is independent from $t^{(i+3)}$.

We can resolve these conflicts in OpenMP through the *depend* clauses with the following relations for all $i \in \{1, \cdots, n_r\}$

$$t^{(i)} \longrightarrow \begin{cases} \emptyset, & \text{if } \exists k \in \mathbb{N}_0 \text{ s.t. } i = 3k+1, \\ t^{(i-1)},\ t^{(i+2)}, & \text{else.} \end{cases} \tag{56}$$

Through the dependency mechanism in OpenMP, these relations define a directed graph which automatically prevents conflicts when applying the matrix A. Taking the example of a grid with nine divisions in the r direction, Figure 7 illustrates the relation (56), i.e. the dependency graph between tasks. In summary, the tasks $t^{(3k+1)}$ are can be run right from the start because they are independent from all other tasks. The execution continues with the tasks $t^{(3k+2)}$, then with the tasks $t^{(3k+3)}$. We then have a theoretical maximum speedup of around $n_r/3$ which ensures a good scaling for large problems.

6.3 Smoothing

As in the Section 5.5, we define the first line for the circle and radial partial smoother as white.

Given the change from circle to radial smoothing of (26), the color of the outermost circle smoother line will be defined. However, for large grids, the difference in smoothing between two

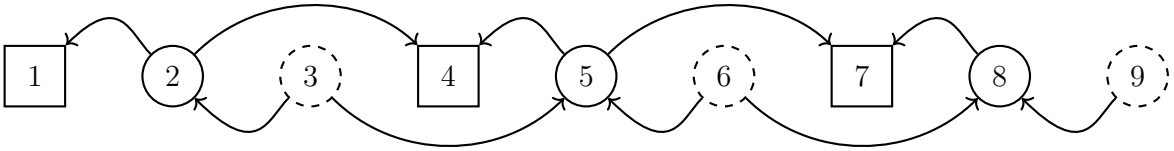


Fig. 7: Dependency graph for the application of A. Each circle/square is a task, i.e., the handling of one line of nodes. Tasks $t^{(3k+1)}$, $k = 0, 1, 2, \ldots$ are within a square and are independent from all other tasks. An arrow from task $t^{(i)}$ to task $t^{(j)}$, indicates that task $t^{(i)}$ must wait for task $t^{(j)}$ to finish.

lines is minimal – so, to simplify the notation, we consider that the outermost line of the circle partial smoother is white too. Based on the parallelism implemented for the application of A, we can also parallelize the construction of the matrices $A_{s_c s_c}$ during the setup. Here, we focus on the parallelization of the complete smoother, i.e., applying all four of the partial smoothers s_c ($s \in \{C, R\},\ c \in \{B, W\}$) in a block Gauss-Seidel approach.

Let's remind Equation (30), i.e.

$$A^{(i)}_{s_c s_c} u^{(i)}_{s_c} = \left(f^{(i)}_{s_c} - A^{(i)}_{s_c s_c^\perp} u^{(i)}_{s_c^\perp} \right),$$

showing that the smoothing procedure implies for each line the application of the operator $A^{(i)}_{s_c s_c^\perp}$ and the solution of the system itself. For these, we define the tasks $t^{(i)}_{s_c s_c^\perp}$ and $t^{(i)}_{s_c s_c}$, respectively. As before, in the case of the circle partial smoother, a line is a circle with constant radius, and for the radial partial smoother, a line consists of the nodes with a constant angle. For reasons outlined below, we number the lines in increasing order of the angle for the radial partial smoother, i.e. indices $i = 1, \ldots, n_\theta$ correspond to angles $\theta_1, \ldots, \theta_{n_\theta}$, but number the lines in decreasing order for the circle partial smoother, i.e. indices $i = 1, \ldots, n_\mu$ correspond to radii $r_{n_\mu}, \ldots, r_{n_1}$.

As detailed in Section 5.3, to use an A-give approach for the application of the $A_{s_c s_c^\perp}$ matrices in the smoother, it is required to loop twice over all nodes, once per color of each smoother. Also, the first circle in the domain corresponding to the radial partial smoother, i.e., the line with radius $r_{n_\mu+1}$, must be handled in the circle partial smoother in order to fully update the r_{n_μ} line. This line is classically called a *halo*, and linked to the task $t^{(0)}_{C_W C_W^\perp}$. Thus, there are $n_\mu + 1$ tasks of type $t^{(i)}_{C_W C_W^\perp}$, n_μ tasks of type $t^{(i)}_{C_B C_B^\perp}$, and finally n_θ of type $t^{(i)}_{R_W R_W^\perp}$ (resp. $t^{(i)}_{R_B R_B^\perp}$). However, the application of the specific $(A^{(i)}_{s_c s_c})^{-1}$ matrix is only required for the line i and not for the neighboring ones. Thus, there are approximately $n_\mu/2$ and $n_\theta/2$ tasks of type $t^{(i)}_{s_c s_c}$ for the circle and radial partial smoothers. For these tasks, we keep the numbering of the complete circle (or radial) partial smoothers, e.g. task $t^{(i)}_{R_c R_c}$ corresponds to the i-th line of the radial partial smoother which is a line with constant angle θ_i.

At each relaxation iteration, it is then possible to run in parallel the application of all tasks, for all smoothing types, as long as we respect the following dependencies:

1. Similarly to the application of the matrix A, there are conflicts from concurrent updates between lines of the same type of s_c smoother. These conflicts can be prevented using dependencies between tasks with the rules

$$t^{(i)}_{s_c s_c^\perp} \longrightarrow \begin{cases} \emptyset, & \text{if } \exists k \text{ s.t. } i = 3k+1, \\ t^{(i-1)}_{s_c s_c^\perp},\ t^{(i+2)}_{s_c s_c^\perp}, & \text{else.} \end{cases} \tag{57}$$

where $i \in \{1, \cdots, n_\mu\}$ for the circle smoother, and $i \in \{1, \cdots, n_\theta\}$ for the radial smoother.

2. Naturally, $(A^{(i)}_{s_c s_c})^{-1}$ is applied on $f^{(i)}_{s_c} - A^{(i)}_{s_c s_c^\perp} u^{(i)}_{s_c^\perp}$, and thus depends on the update of $A^{(i)}_{s_c s_c^\perp}$ for this line and the two neighboring lines. Thus, we impose the rule

$$t^{(i)}_{s_c s_c} \longrightarrow t^{(i-1)}_{s_c s_c^\perp},\ t^{(i)}_{s_c s_c^\perp},\ t^{(i+1)}_{s_c s_c^\perp} \tag{58}$$

where $i \in \{1, \cdots, n_\mu\}$ for the circle smoother, and $i \in \{1, \cdots, n_\theta\}$ for the radial smoother.

3. Then, the application of a black smoother for a line must wait for the application of the corresponding white smoother on the neighboring lines, i.e.

$$t^{(i)}_{s_B s_B^\perp} \longrightarrow \begin{cases} t^{(i)}_{s_W s_W}, & \text{if } i \text{ is a white line,} \\ t^{(i-1)}_{s_W s_W},\ t^{(i+1)}_{s_W s_W}, & \text{else.} \end{cases} \tag{59}$$

where $i \in \{1, \cdots, n_\mu\}$ for the circle smoother, and $i \in \{1, \cdots, n_\theta\}$ for the radial smoother.

4. Finally, there are also conflicts from the updates between the circle and radial smoothers. In fact, to start the smoothing of radial lines, the outermost line n_μ of the circle smoother must be finished. Thus, considering that the line n_μ is white and corresponds to task $t^{(1)}_{C_W C_W}$, we impose the additional dependency

$$\forall i \in \{1, \cdots, n_\theta\}, t^{(i)}_{R_c R_c^\perp} \longrightarrow t^{(1)}_{C_W C_W}, \tag{60}$$

In order to start the radial partial smoother as soon as possible, in parallel with the circle partial smoother, we need to finish task $t^{(1)}_{C_W C_W}$ first which is dependent on $t^{(1)}_{C_W C_W^\perp}$, $t^{(2)}_{C_W C_W^\perp}$ and $t^{(0)}_{C_W C_W^\perp}$. Thus, we choose to start the relaxation by sequentially performing the three latter tasks, in parallel with all independent tasks $t^{(3k+1)}_{s_c s_c^\perp}$, which explains the reverse numbering of tasks for the circle partial smoother.

These dependencies are illustrated in Figure 8 on an example with five circle lines, and six radial lines. Note that we do not include the dependencies corresponding to the first rule above, since they follow the same rule as for the application of A but in a descending order, see Figure 7.

By accumulating all of these rules through the OpenMP dependencies, we obtain a directed graph of tasks which prevents any conflicts from happening. Similarly to the application of A, we then have an asymptotic maximum speed-up of $\lfloor n_\mu/3 \rfloor + \lfloor n_\theta/3 \rfloor$, also ensuring a good scaling for large problem sizes.

7 Numerical experiments

In the following, we will present numerical results for the model problem (1). The coefficients α and β used here as well as the manufactured solution are obtained from the use cases in [Zon19], [KKR22] and, precisely, [BLK+23, Eq. (18)-(19) and (23)]. These cases are inspired by the simulation of plasma in tokamak fusion reactors. Here, we consider a *Cartesian solution* with oscillations aligned with the Cartesian grid:

$$u(x, y) = C \left(1 + \frac{r(x, y)}{R_{max}}\right)^6 \left(1 - \frac{r(x, y)}{R_{max}}\right)^6 \cos(2\pi x) \sin(2\pi y). \tag{61}$$

Figure 9 illustrate the solution for the Shafranov and Czarny geometries.

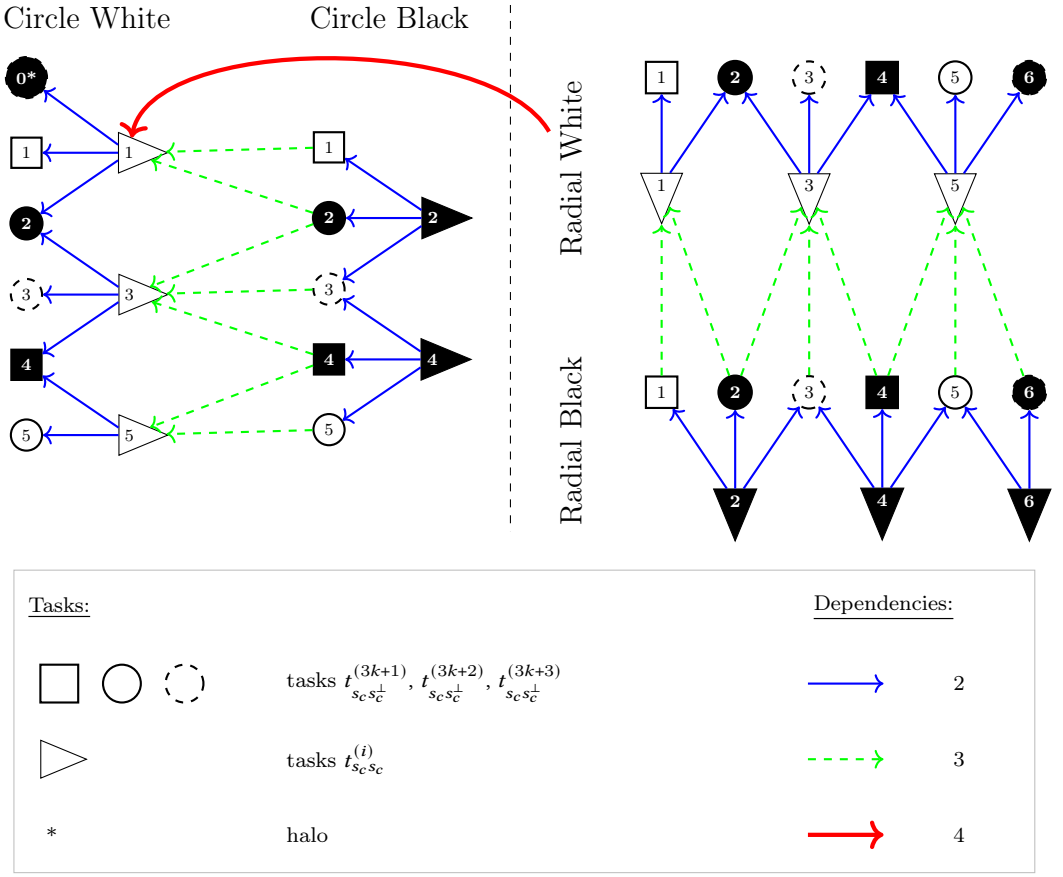


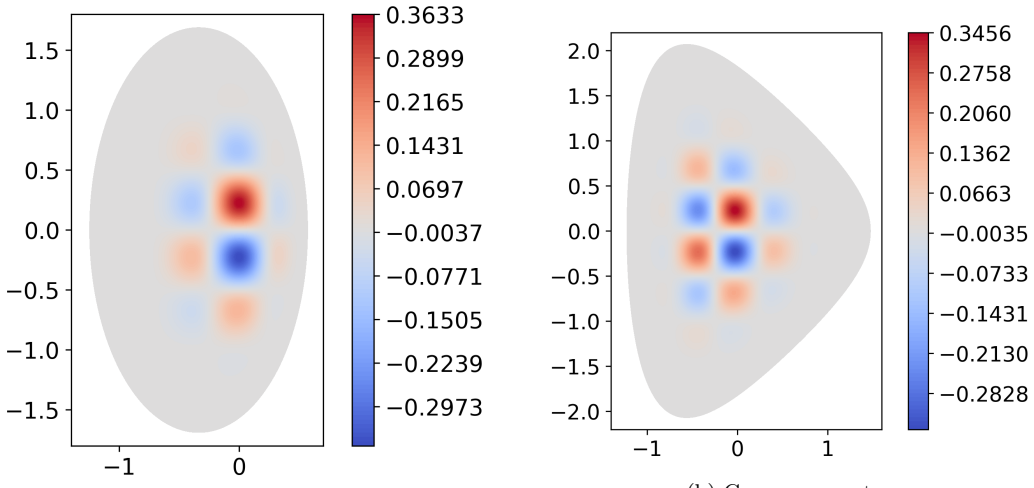
Fig. 8: Dependency graph for the application of the smoother. Each node is a task, i.e., the handling of a circle line of nodes. The tasks $t^{(i)}_{s_c s_c^\perp}$, $i = 2, 3, 5$ and $i = 2, 3, 5, 6$, are represented by circles and tasks $t^{(i)}_{s_c s_c^\perp}$, $i = 1, 4$ are squares. Tasks $t^{(i)}_{s_c s_c}$, $i = 1, 2, \ldots$, are represented by triangles. The different type of dependencies are colored: 2=blue, 3=green, and 4=red. For the sake of a simple presentation, we do not display the dependencies from (57), as they are identical to what is shown in Figure 7.

For the coefficients α and β, we set

$$\alpha(r) = \exp\left[-\tanh\left(\frac{\frac{r(x,y)}{R_{max}} - r_p}{\delta_r}\right)\right], \tag{62}$$

$$\beta(r) = \exp\left[\tanh\left(\frac{\frac{r(x,y)}{R_{max}} - r_p}{\delta_r}\right)\right], \tag{63}$$

where $\delta_r = 0.05$ and $r_p = 0.7$, as in [BLK+23], and $R_{max} = 1.3$ by our own choice as in [KKR22]. Note that R_{max} was set to 1 in [BLK+23]. See Figure 10 for an illustration of the coefficients α and β.



(a) Shafranov geometry

(b) Czarny geometry

Fig. 9: Illustration of the manufactured solution (61) for the Shafranov and Czarny geometries.

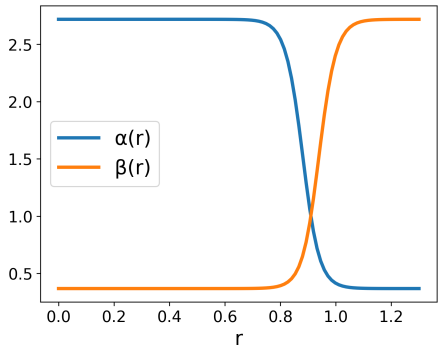


Fig. 10: Coefficients α (62) and β (63) used in Equation (1).

The solver GMGPolar was initially developed for the Equation (1) with $\beta = 0$. Here, we use a more general form to show that GMGPolar naturally extends to a nontrivial β coefficient, exhibiting similar properties of convergence and accuracy as in the simpler case.

To handle the singularity introduced by the polar geometry at $r_1 = 0$, GMGPolar has demonstrated in [KKR22] the possibility through special considerations of the finite difference discretizations. However, we simplify these considerations by simply setting a minimum radius $r_1 = 1e-8$ and constructing Dirichlet boundary conditions on this innermost circle of the domain from the manufactured solution. When considering the asymptotic computational and memory complexities, we did not consider the boundary conditions, so this choice for r_1 does not affect this study.

In our experiments, the iterations are stopped based on a relative reduction threshold of $1e-11$ for the residual, or after a maximum of 300 iterations of GMGPolar. At each level of the multigrid cycle, we perform one pre- and one post-smoothing step. We always use implicit extrapolation and the matrix-free implementation of GMGPolar.

Table 4: Numerical scalability of GMGPolar from 787k to 50m degrees of freedom on the Shafranov geometry and Cartesian solution (61).

$n_r \times n_\theta$	its	$\|err\|_{\ell_2}$	ord.	$\|err\|_\infty$	ord.
769 × 1 024	70	3.18e-08	–	9.40e-07	-
1 537 × 2 048	67	2.86e-09	3.48	1.18e-07	3.00
3 073 × 4 096	64	2.55e-10	3.49	1.47e-08	3.00
6 145 × 8 192	61	2.26e-11	3.49	1.84e-09	3.00

Table 5: Numerical scalability of GMGPolar from 787k to 50m degrees of freedom on the Czarny geometry and the Cartesian solution (61).

$n_r \times n_\theta$	its	$\|err\|_{\ell_2}$	ord.	$\|err\|_\infty$	ord.
769 × 1 024	56	2.31e-08	-	7.05e-07	-
1 537 × 2 048	54	2.01e-09	3.51	8.81e-08	3.00
3 073 × 4 096	51	1.79e-10	3.50	1.10e-08	3.00
6 145 × 8 192	49	1.58e-11	3.50	1.38e-09	3.00

Our experiments are performed using an OpenMP parallelization on a single node with 2-sockets having 256 GB memory. Each socket is an AMD EPYC 7702 "Rome" processor at 2GHz containing 64 cores, constituting a separate NUMA (non-uniform memory access) domain. Here, we fix the CPU frequency to 1.8 Ghz and only use one thread per core. We use LIKWID 5.2.2 [HWT10, GEHW22] for measuring the amount of computations in flops, and the memory consumption in Mbytes. Finally, to solve the problem on the coarsest grid of GMGPolar, we use MUMPS v5.4.1 [AAB+15].

In this section, we show the different types of scalability for GMGPolar from numerical through weak and finally strong scaling.

7.1 Numerical scalability of GMGPolar

In this section, we first demonstrate numerical scalability for the model problem and the geometries considered. Note thatn numerical scalability of GMGPolar for other configurations and smaller model problems has already been shown in [KKR22].

Here, we consider the right hand side given by the manufactured solution (61), called the *Cartesian solution*. The problem is defined on the Shafranov geometry, see Figure 9a, and the Czarny geometry, see Figure 9b. We construct grids of increasing refinement to get problems of size from $8e5$ to $5e7$ DOFs.

Tables 4 and 5 provide the errors in (weighted) ℓ_2 and ∞ norm with respect to the manufactured solution, and provide the error reduction order as *ord.* for both norms. The order is calculated as the error reduction from one row to its predecessor, i.e.,

$$\text{ord} = \log\left(\frac{\|err_k\|}{\|err_{k-1}\|}\right) / \log\left(\sqrt{\frac{gridsize_{k-1}}{gridsize_k}}\right).$$

For both geometries, we obtain an approximation order around 3.5 (which is roughly below 4, which we obtained for the polar use case) in ℓ_2-norm as expected from the use of the implicit extrapolation scheme. In fact, we obtain similar approximation orders as in [KKR21] and [KKR22] but solving here the equation (1) with a non-zero β coefficient and also the Czarny geometry. Note that the number of iterations in all cases stays relatively stable with respect to the problem size. In fact, the number of iterations even tends to decrease for larger problems, e.g. from 56 to 49 iterations from the smallest to the largest problems on the Czarny geometry.

7.2 Weak scalability of GMGPolar

We now provide the weak scaling experiment for the exact same problems of the previous sections, starting from $5e7$ degrees of freedom (DOFs). We demonstrate the overall parallel efficiency of GMGPolar in shared memory parallelism.

In Figure 11 we give the amount of computation in flops for the multigrid iterations of GMGPolar depending on the problem size, as measured using likwid. We include the asymptotic theoretical number of flops. The theoretical asymptotic computational complexity is computed based on Eq. (55) with the actual number of iterations, completed by the results in Table 3 and the specifities of the implicit extrapolation detailed in Section 5.5.

As expected, we observe that the complexity of GMGPolar is increasing linearly with the size of the problem. Proving this is one of the main results from this article. The computational cost measured in practice is twice as large as the theoretical one, which leaves space for optimization. However, note that the expected theoretical numbers are asymptotic and that lower order computations are not considered in the theoretical sum.

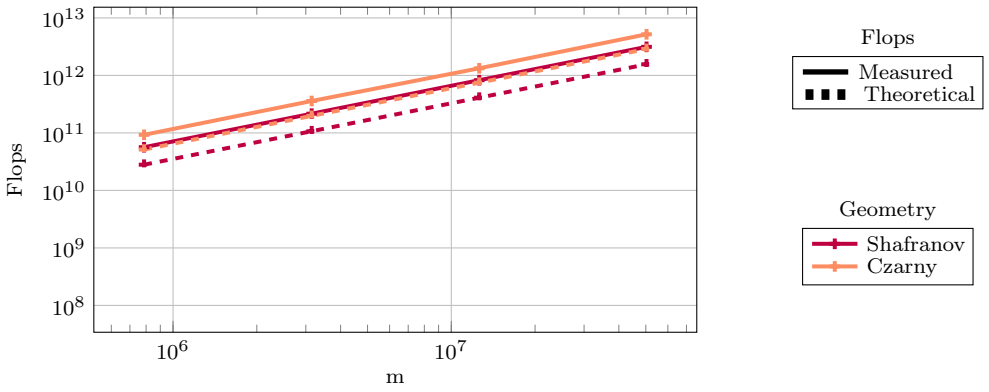


Fig. 11: Computational cost in flops when applying GMGPolar for the manufactured Cartesian solution (61) on the geometries Shafranov and Czarny. We include the number of flops as measured by LIKWID as well as the theoretical (asymptotic) expectations.

Finally, we perform a weak scalability experiment with the parallel application of GMGPolar on these same problems of size up to $5e7$ DOFs, from 1 to 64 OpenMP threads. Here, each thread handles a constant number of $5e7$ DOFs, and the ideal behavior is obtained when the execution time stays identical for all problem sizes. The setup phase of GMGPolar is (currently) sequential, thus we focus again on the parallel execution of the multigrid iterations. Figure 12

gives the parallel efficiency of the multigrid cycle phase depending on the problem size. We observe an excellent efficiency from 1 to 16 with 71% and 84% respectively for the Shafranov and Czarny geometries. This parallel efficiency is obtained thanks to the efficient parallelization scheme proposed in Section 6, using task-based parallelism. Using the full socket of 64 cores still provides additional gain overall, with an efficiency staying around 50% in the case of Czarny, although there is also potential for future improvement. In fact, the total computation time only doubles while the problem size is multiplied by 64. Note that GMGPolar seems slightly more scalable in the case of the Czarny geometry, it is likely that with the more expensive and repetitive computation of the mapping of the geometry in the case of Czarny, the granularity of the problem increases and this in turn improves the parallel efficiency of the implementation. In the next section, we will detail the scalability of each parts of the multigrid cycle to identify the ones currently limiting our parallel implementation.

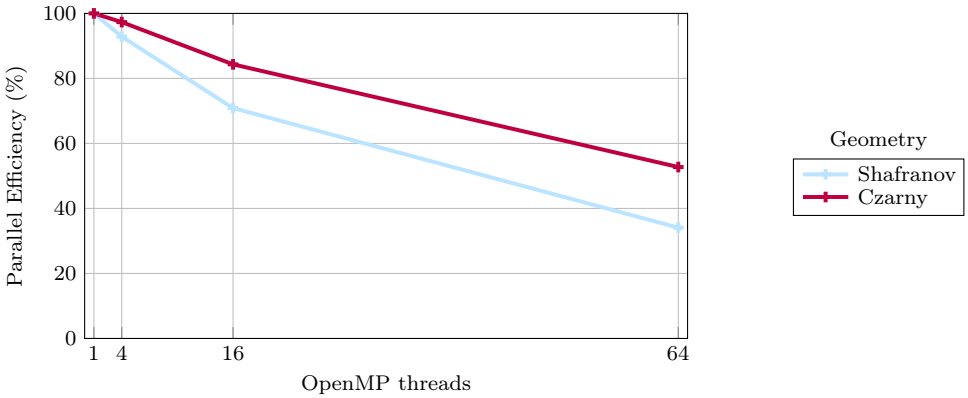


Fig. 12: **Weak scaling of GMGPolar.** Parallel efficiency compared to the sequential run from 1 to 64 cores on grids of size 769×1024 to 6145×8192, manufactured Cartesian solution (61).

7.3 Strong scalability of GMGPolar

In this section, we focus on a finer discretizations of Eq. (1) for which we consider again the two different geometries and the Cartesian solution (61). First, we perform a strong scaling experiment by applying GMGPolar with 1 up to 64 OpenMP threads, on a grid of size 6145×8192, i.e. a grid with about 50 million DOFs. Figure 13 presents the resulting execution times for the multigrid cycle phase respectively for the Shafranov and Czarny geometries. Again, the setup phase is excluded since it is sequential in our implementation. Additionally to the complete multigrid iterations, we detail the efficiency of several inner steps: the smoothing, the computation of the residual, the computation of the finest residual, the restriction and prolongation, the solution of the coarse problem.

From Figure 13, we observe a very good overall scaling from 1 to 16 cores, where we get a good speed-up of around 13. Although using the full socket of 64 cores still decreases the execution time, the scalability decreases and we see there is potential for future improvement for the other phases. The most efficient step is the smoothing which is often viewed as the critical part in a multigrid method. In fact, up to 32 threads, the smoothing procedure shows a great scalability. Then, the performance of the smoothing decreases but we still get a speed-up of

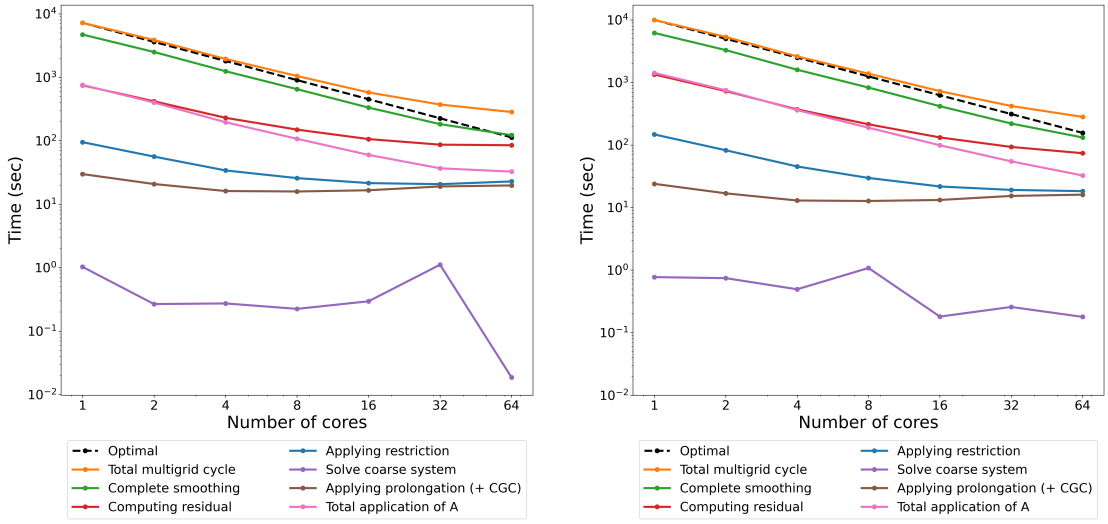


Fig. 13: **Strong scaling of GMGPolar.** From 1 to 64 cores on a grid of size 6145×8192, Shafranov geometry Figure 2a (left) and Czarny geometry Figure 2b (right) and manufactured solution (61).

around 35 when using 64 threads. This good scalability comes from the implemented task-based parallelism, making use of the decomposition in lines of fixed radius and the limited interactions between these.

Most of the other operations in GMGPolar have a minor effect on the total run time. For example, the matrix-free application of the residual and prolongation operators do not scale but have a low impact on the total execution time. Only the computation of the finest residual significantly worsens the impact scalability and needs to be improved for future applications. In fact, with 1 thread, computing the residual amounts to around 10% of the total execution time, and increases to 30% with 64 threads.

8 Conclusion

In this paper, we have presented a matrix-free approach of GMGPolar which uses a nine-point stencil inside a taylored, implicitly extrapolated multigrid scheme with zebra line smoothing adapted to different parts of the domain. We have newly presented a rigorous analysis of flops and memory usage and demonstrated optimal, linear asymptotic, complexity of the considered solver in both computations and memory. Although the solution of the coarsest grid problem does not have a linear complexity, considering a sufficiently large number of grid levels, it can be neglected for the overall cost analysis of the GMGPolar solver. Our rigorous analysis is kept as general as possible such that its framework ideas can be transferred to matrix-free approaches of other stencil-based operators.

Furthermore, we have presented a task-based multithreaded parallelism for GMGPolar which takes advantage of the disk-shaped geometry of the problem and defines dependencies accordingly. We have demonstrated numerical, weak, and strong scalability for use cases from fusion plasma applications. GMGPolar can now sufficiently handle disk-like geometries with tens and hundreds of millions of degrees of freedom. Hereby, the implemented node-level parallelism scales very well

from 1 to a 16, 32, or 64-fold of cores. Together with the results of [BLK+23], the solver is ready to be coupled with gyrokinetic codes such as Gysela [GAB+16].

Future developments of GMGPolar will involve further optimizations of the run-time and the use of accelerators for computation.

Declarations

Funding and Acknowledgements
We acknowledge the computing time granted on the HPC cluster CARO. Furthermore, we would like to thank Nils Kanning for support on CARO and Melven Röhrig-Zöllner and Johannes Wendler for fruitful hints and discussions on LIKWID and performance monitoring issues.

Conflicts of interest/Competing interests
None.

Availability of data and material
No particular data was used.

Code availability
The code is available on GitHub https://github.com/SciCompMod/GMGPolar and with tagged version 1.0.2 and persistent identifier on zenodo https://doi.org/10.5281/zenodo.1040929. Grids have been created with version 1.0.0, simulations have been performed with version 1.0.2.

Authors' contributions
MJK, CK, and UR developed the numerical algorithm, the implementation was done by PL, CS, MJK and CK. MJK and CK did the initial implementation in Matlab, PL and CS ported the code to C/C++ and did the task-based parallelization. Simulations were conducted by MJK. All authors wrote and revised the script.

References

AAB+15. Patrick Amestoy, Cleve Ashcraft, Olivier Boiteau, Alfredo Buttari, Jean-Yves L'Excellent, and Clément Weisbecker. Improving multifrontal methods by means of block low-rank representations. *SIAM Journal on Scientific Computing*, 37(3):A1451–A1474, 2015.

Bar88. Saulo RM Barros. The Poisson equation on the unit disk: a multigrid solver using polar coordinates. *Applied Mathematics and Computation*, 25(2):123–135, 1988.

BBG+18. Nicolas Bouzat, Camilla Bressan, Virginie Grandgirard, Guillaume Latu, and Michel Mehrenberger. Targeting realistic geometry in Tokamak code Gysela. *ESAIM: Proceedings and Surveys*, 63:179–207, 2018.

Ber97. Klaus Bernert. τ-extrapolation—theoretical foundation, numerical experiment, and application to navier–stokes equations. *SIAM Journal on Scientific Computing*, 18(2):460–478, 1997.

BL11. Achi Brandt and Oren E Livne. *Multigrid Techniques: 1984 Guide with Applications to Fluid Dynamics, Revised Edition*. SIAM, 2011.

BLK+23. Emily Bourne, Philippe Leleux, Katharina Kormann, Carola Kruse, Virginie Grandgirard, Yaman Güclü, Martin J. Kühn, Ulrich Rüde, Eric Sonnendrücker, and Edoardo Zoni. Solver comparison for poisson-like equations on tokamak geometries. *Journal of Computational Physics*, 488:112249, 2023.

CH08. Olivier Czarny and Guido Huysmans. Bézier surfaces and finite elements for mhd simulations. *Journal of computational physics*, 227(16):7423–7445, 2008.

GAB+16. Virginie Grandgirard, Jérémie Abiteboul, Julien Bigot, Thomas Cartier-Michaud, Nicolas Crouseilles, Guilhem Dif-Pradalier, Ch Ehrlacher, Damien Esteve, Xavier Garbet, Ph Ghendrih, et al. A 5D gyrokinetic full-f global semi-Lagrangian code for flux-driven ion turbulence simulations. *Computer Physics Communications*, 207:35–68, 2016.

GEHW22. Thomas Gruber, Jan Eitzinger, Georg Hager, and Gerhard Wellein. LIKWID v5.2.2, August 2022. https://zenodo.org/records/6980692.

HHP+21. Matthias Hoelzl, Guido Huijsmans, Stanislas Pamela, Marina Becoulet, Eric Nardon, Francisco Javier Artola, Boniface Nkonga, Calin Atanasiu, Vinodh Bandaru, Ashish Bhole, et al. The JOREK non-linear extended MHD code and applications to large-scale instabilities and their control in magnetically confined fusion plasmas. *Nuclear Fusion*, 2021.

HWT10. G. Hager, G. Wellein, and J. Treibig. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *2012 41st International Conference on Parallel Processing Workshops*, pages 207–216, Los Alamitos, CA, USA, sep 2010. IEEE Computer Society.

JR96. Michael Jung and Ulrich Rüde. Implicit extrapolation methods for multilevel finite element computations. *SIAM Journal on Scientific Computing*, 17(1):156–179, 1996.

JR98. Michael Jung and Ulrich Rüde. Implicit extrapolation methods for variable coefficient problems. *SIAM Journal on Scientific Computing*, 19(4):1109–1124, 1998.

KKR21. Martin Joachim Kühn, Carola Kruse, and Ulrich Rüde. Energy-minimizing, symmetric discretizations for anisotropic meshes and energy functional extrapolation. *SIAM Journal on Scientific Computing*, 43(4):A2448–A2473, 2021.

KKR22. Martin J Kühn, Carola Kruse, and Ulrich Rüde. Implicitly extrapolated geometric multigrid on disk-like domains for the gyrokinetic Poisson equation from fusion plasma applications. *Journal of Scientific Computing*, 91(1):1–27, 2022.

LKS+23. Philippe Leleux, Martin J. Kühn, Christina Schwarz, Carola Kruse, Ulrich Rüde, and Allan Kuhn. GMGPolar v1.02, December 2023. https://doi.org/10.5281/zenodo.10409297.

Rüd91. Ulrich Rüde. *Extrapolation and related techniques for solving elliptic equations*. Citeseer, 1991.

Sch21. Christina Schwarz. Geometric multigrid for the Gyrokinetic Poisson equation from fusion plasma applications, 2021. Universität Erlangen-Nürnberg. https://elib.dlr.de/146684/.

TOS01. Ulrich Trottenberg, Cornelius W Oosterlee, and Anton Schüller. *Multigrid*. Academic Press, London San Diego, 2001.

WC11. John Wesson and David J Campbell. *Tokamaks*, volume 149. Oxford university press, 2011.

ZG19. Edoardo Zoni and Yaman Güçlü. Solving hyperbolic-elliptic problems on singular mapped disk-like domains with the method of characteristics and spline finite elements. *Journal of Computational Physics*, 398:108889, 2019.

Zon19. Edoardo Zoni. *Theoretical and numerical studies of gyrokinetic models for shaped Tokamak plasmas*. PhD thesis, Technische Universität München, 2019.

Appendix A: Detailed computational complexity and analysis framework

In this appendix, we detail how to compute the complexity of the components constituting GMGPolar. The main steps and results of this appendix are given in Section 5, and we use the same notation as introduced in Section 5.1. We remind the reader that $\cos\theta$, $\sin\theta$, α, and β are precomputed with a negligible cost compared to a linear complexity. While Appendix 8.1 provides exact number of flops for the geometries considered, Appendix 8.2 is kept general for applications with mappings on curvilinear coordinates.

8.1 Jacobian of the mapping

Note that the algorithms we provide to compute the Jacobian are far from optimal. To stay general with respect to the application, the corresponding codes are generated from the analytical formula of the mapping function using the python library SymPy[2] for symbolic mathematics. Appendix 8.3 is written in general framework form so that it can be reused for other stencil-based operators and Appendix 8.4 is applicable for circle-radial line smoothers (also providing the extrapolation case).

In order to compute the entries of the matrix using the nine point stencil as introduced in Section 3, the Jacobian of the geometry mapping $DF := \begin{pmatrix} J_{rr} & J_{r\theta} \\ J_{\theta r} & J_{\theta\theta} \end{pmatrix}$ is required. This Jacobian naturally depends on the choice of the geometry. Based on Section 2, we can use:

1. The polar geometry with the Jacobian

$$DF_{polar} = \frac{1}{R_{max}} \begin{pmatrix} \cos\theta & -r\sin\theta \\ \sin\theta & r\cos\theta \end{pmatrix}, \tag{64}$$

which can be computed with $\mathfrak{F}_m(DF_{polar}) = 6$ flops based on Algo. 3.

Algorithm 3 DF_{polar}: `(2 mult., 4 div. = 6 flops)`

Input: $(r,\theta), (\cos\theta, \sin\theta), Rmax$.
1: $J_{rr} = \cos\theta / R_{max}$ `(1 div.)`
2: $J_{r\theta} = -r\sin\theta / R_{max}$ `(1 mult., 1 div.)`
3: $J_{\theta r} = \sin\theta / R_{max}$ `(1 div.)`
4: $J_{\theta\theta} = r\cos\theta / R_{max}$ `(1 mult., 1 div.)`

2. The Shafranov geometry (3) with the Jacobian

$$DF_{shafranov} = \frac{1}{R_{max}} \begin{pmatrix} -(\kappa-1)\cos\theta - 2\delta\frac{r}{R_{max}} & r(\kappa-1)\sin\theta \\ (\kappa+1)\sin\theta & r(\kappa+1)\cos\theta \end{pmatrix} \tag{65}$$

which can be computed with $\mathfrak{F}_m(DF_{shafranov}) = 18$ flops based on Algo. 4.

3. The Czarny geometry (4) with the Jacobian

$$DF_{czarny} = \begin{pmatrix} -\cos\theta/\zeta & r\sin\theta/\zeta \\ \frac{\lambda\xi\sin\theta}{2-\zeta}\left(1 + \frac{\varepsilon r\cos\theta}{(2-\zeta)\zeta}\right) & \frac{\lambda\xi r}{2-\zeta}\left(\cos\theta - \frac{\varepsilon r\sin\theta^2}{(2-\zeta)\zeta}\right) \end{pmatrix} \tag{66}$$

with $\zeta = \sqrt{\varepsilon(\varepsilon + 2\frac{r}{R_{max}}\cos\theta) + 1}$ and $\xi = 1/\sqrt{1-\varepsilon^2/4}$, which can be computed with $\mathfrak{F}_m(DF_{czarny}) = 112$ flops based on Algo. 5.

[2] https://www.sympy.org/en/index.html

Algorithm 4 $DF_{shafranov}$: `(8 mult., 5 add, 5 div. = 18 flops)`

Input: $(r,\theta),(\cos\theta,\sin\theta),(\delta,\kappa),R_{max}$.

1: $J_{rr} = (-2\delta\frac{r}{R_{max}} - \kappa\cos\theta + \cos\theta)/R_{max}$ `(3 mult., 2 add, 2 div.)`

2: $J_{r\theta} = \frac{r}{R_{max}}(\kappa\sin\theta - \sin\theta)$ `(2 mult., 1 add, 1 div.)`

3: $J_{\theta r} = (\kappa+1)\sin\theta/R_{max}$ `(1 mult., 1 add, 1 div.)`

4: $J_{\theta\theta} = \frac{r}{R_{max}}(\kappa\cos\theta + \cos\theta)$ `(2 mult., 1 add, 1 div.)`

Algorithm 5 DF_{Czarny}: `(52 mult., 25 add, 23 div., 12 sqrt = 112 flops)`

Input: $(r,\theta),(\cos\theta,\sin\theta),(\varepsilon,\lambda),R_{max}$.

1: $J_{rr} = \left(\frac{-\cos\theta}{\sqrt{\varepsilon(\varepsilon+2\frac{r}{R_{max}}\cos\theta)+1}}\right)/R_{max}$ `(3 mult., 2 add, 3 div., 1 sqrt)`

2: $J_{r\theta} = r\left(\frac{\sin\theta}{\sqrt{\varepsilon(\varepsilon+2\frac{r}{R_{max}}\cos\theta)+1}}\right)/R_{max}$ `(4 mult., 2 add, 3 div., 1 sqrt)`

3: $J_{\theta r} = \left(\frac{\lambda\varepsilon\frac{r}{R_{max}}\sin\theta\cos\theta}{\sqrt{1-(1/4)\varepsilon^2}\left(2-\sqrt{\varepsilon(\varepsilon+2\frac{r}{R_{max}}\cos\theta)+1}\right)^2\sqrt{\varepsilon(\varepsilon+2\frac{r}{R_{max}}\cos\theta)+1}} + \frac{\lambda\sin\theta}{\sqrt{1-(1/4)\varepsilon^2}\left(2-\sqrt{\varepsilon(\varepsilon+2\frac{r}{R_{max}}\cos\theta)+1}\right)}\right)/R_{max}$
`(22 mult., 11 add, 9 div., 5 sqrt)`

4: $J_{\theta\theta} = r\left(\frac{-\lambda\varepsilon\frac{r}{R_{max}}\sin^2\theta}{\sqrt{1-(1/4)\varepsilon^2}\left(2-\sqrt{\varepsilon(\varepsilon+2\frac{r}{R_{max}}\cos\theta)+1}\right)^2\sqrt{\varepsilon(\varepsilon+2\frac{r}{R_{max}}\cos\theta)+1}} + \frac{\lambda\cos\theta}{\sqrt{1-(1/4)\varepsilon^2}\left(2-\sqrt{\varepsilon(\varepsilon+2\frac{r}{R_{max}}\cos\theta)+1}\right)}\right)/R_{max}$
`(23 mult., 11 add, 9 div., 5 sqrt)`

8.2 functions a^{xx}

Using Equation (10), the functions a^{xx} are defined as

$$\begin{aligned}
&\begin{pmatrix} a^{rr} & \frac{1}{2}a^{r\theta} \\ \frac{1}{2}a^{r\theta} & a^{\theta\theta} \end{pmatrix} = \frac{1}{2}\alpha DF^{-1}DF^{-T}|\det(DF)| \\
\iff &\begin{cases} a^{rr} = \frac{1}{2}\alpha\left(J_{\theta\theta}^2 + J_{r\theta}^2\right)/|\det(DF)|, \\ a^{\theta\theta} = \frac{1}{2}\alpha\left(J_{\theta r}^2 + J_{rr}^2\right)/|\det(DF)|, \\ a^{r\theta} = \alpha\left(J_{\theta\theta}J_{\theta r} + J_{r\theta}J_{rr}\right)/|\det(DF)|, \end{cases}
\end{aligned} \tag{67}$$

with $\det(DF) = J_{rr}J_{\theta\theta} - J_{r\theta}J_{\theta r}$.

Since the functions a^{xx} are built from the same elements, it is more efficient to compute all three functions simultaneously with the cost $\mathfrak{F}_m(a^{xx}) = 19 + \mathfrak{F}_m(DF)$, as detailed in Algo. 6.

In order to build/apply A or the smoother matrices $A_{s_cs_c}$ and $A_{s_cs_c^\perp}$, different combinations of the functions a^{xx} must be computed with a corresponding computational cost. In fact, considering Algo. 6, there is a common cost of 6 flops for all three functions (lines 1 to 4) plus 4 flops, 4 flops, and 5 flops, respectively to compute a^{rr}, $a^{\theta\theta}$, and $a^{r\theta}$.

8.3 Matrix-free application of A

We now detail the computation of the complexity for the construction and the matrix-free application of the matrix A. We recall that the matrix is based on the nine point finite-difference discretization introduced in Section 3 with the entries from Equations (19)–(21).

In this section, we establish some tools in order to compute the computational cost per node required for the construction of a matrix in the context of discretized PDEs. We believe that our method can be generalized to most approaches (using e.g. other finite difference stencils or finite

Algorithm 6 a^{xx} computation (DF, 13 mult., 4 add, 1 div., 1 abs. = 19 + $\mathfrak{F}_m(DF)$)

1: Compute $DF = \begin{pmatrix} J_{rr} & J_{r\theta} \\ J_{\theta r} & J_{\theta\theta} \end{pmatrix}$ ($\mathfrak{F}_m(DF)$)		
2: $detDF = J_{rr}J_{\theta\theta} - J_{r\theta}J_{\theta r}$ (2 mult., 1 add)		
3: $absdetDF = \mathrm{abs}(detDF)$ (1 abs.)		= 6 + $\mathfrak{F}_m(DF)$
4: $coeff = 0.5\alpha(r)/absdetDF$ (1 mult., 1 div.)		
// Only for A, $A_{s_c s_c}$, and circle/extrapolation $A_{s_c s_c^\perp}$, otherwise skip next line		
5: $a^{rr} = coeff \times (J_{\theta\theta}J_{\theta\theta} + J_{r\theta}J_{r\theta})$ (3 mult., 1 add)		= 4
// Only for A, $A_{s_c s_c}$, and radial/extrapolation $A_{s_c s_c^\perp}$, otherwise skip next line		
6: $a^{\theta\theta} = coeff \times (J_{\theta r}J_{\theta r} + J_{rr}J_{rr})$ (3 mult., 1 add)		= 4
// Only for A, $A_{s_c s_c^\perp}$, otherwise skip next line		
7: $a^{r\theta} = 2coeff \times (J_{\theta\theta}J_{\theta r} + J_{r\theta}J_{rr})$ (4 mult., 1 add)		= 5

elements) in order to get a complete knowledge of how expensive constructing the discretized operator is. As detailed below, our method is based on decomposing the computation of a stencil in multiple contributions, and visualizing the corresponding costs as a diagram.

Construction of a generic matrix

Let us consider a square matrix M of size m such that its entries can be expressed as the sum of a few contributions, i.e. $M_{ij} = \sum_{k \in K_{ij}} M_{ij}^{(k)}$, where K_{ij} depends on the entry (i, j). Note that the contributions are not uniquely defined and may vary depending on what shall be precomputed and reused.

If we consider that the matrix is built from a discretization using a specific stencil, then the non-zero entries of the matrix M are defined by the form of the stencil, and the sets of contributions K_{ij} can defined by the expression of each update in the stencil.

To explain our general approach, let us now consider the example of the nine-point stencil (19)–(21) used in GMGPolar and a node with coordinate indices (s, t).

In this example, we directly use the coordinate indices (s, t) of each node instead of i or j and we use A instead of M to distinguish the example from the general consideration. We assume A to be ordered with the coordinate indices. This way, the matrix is defined up to a certain renumbering which can be chosen freely upon implementation. Generally, we use $i = s * n_\theta + t$, i.e. numbering nodes following the circles lines. Finally, to make the notation more visual, we replace the coordinates (s, t) by $\otimes$, and coordinates $(s \pm 1, t \pm 1)$ by an arrow indicating the direction of the corresponding node with respect to the node (s, t) (e.g. $\searrow$ for the coordinates $(s - 1, t + 1)$).

We have the expression of each update

$$
\begin{aligned}
\texttt{[top]} \quad A_{\otimes\uparrow} &= -\frac{k_t + k_{t-1}}{h_s}\,\frac{a^{rr}_{\otimes} + a^{rr}_{\uparrow}}{2}, & A_{\otimes\nearrow} &= \frac{a^{r\theta}_{\uparrow} + a^{r\theta}_{\rightarrow}}{4}, \quad \texttt{[top-right]} \\
\texttt{[bottom]} \quad A_{\otimes\downarrow} &= -\frac{k_t + k_{t-1}}{h_{s-1}}\,\frac{a^{rr}_{\downarrow} + a^{rr}_{\otimes}}{2}, & A_{\otimes\nwarrow} &= \frac{a^{r\theta}_{\uparrow} + a^{r\theta}_{\leftarrow}}{4}, \quad \texttt{[top-left]} \\
\texttt{[right]} \quad A_{\otimes\rightarrow} &= -\frac{h_s + h_{s-1}}{k_t}\,\frac{a^{\theta\theta}_{\otimes} + a^{\theta\theta}_{\rightarrow}}{2}, & A_{\otimes\searrow} &= \frac{a^{r\theta}_{\downarrow} + a^{r\theta}_{\rightarrow}}{4}, \quad \texttt{[bottom-right]} \\
\texttt{[left]} \quad A_{\otimes\rightarrow} &= -\frac{h_s + h_{s-1}}{k_{t-1}}\,\frac{a^{\theta\theta}_{\leftarrow} + a^{\theta\theta}_{\otimes}}{2}, & A_{\otimes\swarrow} &= \frac{a^{r\theta}_{\downarrow} + a^{r\theta}_{\leftarrow}}{4}, \quad \texttt{[bottom-left]}.
\end{aligned}
\tag{68}
$$

$$
A_{\otimes\otimes} = -\left[A_{\otimes\uparrow} + A_{\otimes\downarrow} + A_{\otimes\rightarrow} + A_{\otimes\rightarrow}\right] + \frac{(h_s + h_{s-1})(k_t + k_{t-1})}{4}\beta_{s,t}\,|\det DF_{s,t}|. \quad \texttt{[central]}.
$$

Continuing the above example, we get the set of contributions $K_{\otimes\uparrow} = \{1, 2\}$ with

$$\begin{aligned} A_{\otimes\uparrow} &= A^{(1)}_{\otimes\uparrow} + A^{(2)}_{\otimes\uparrow}, \\ A^{(1)}_{\otimes\uparrow} &= -\frac{k_t + k_{t-1}}{h_s}\frac{a^{rr}_{\otimes}}{2}, \\ A^{(2)}_{\otimes\uparrow} &= -\frac{k_t + k_{t-1}}{h_s}\frac{a^{rr}_{\uparrow}}{2}. \end{aligned} \tag{69}$$

From Equation (68), we understand that some elements which are common to different updates should be precomputed to save flops. In our example, the functions a^{xx} are to be computed only once per node (s, t) while the sums and quotients of h and k are cheap to compute. In a general sense, we further write

$$M^{(k)}_{ij} = \gamma^k_{ij} x^k_{ij},$$

where we have certain variables $X = (x^k_{ij})_{ij\in\{1,\dots,m\},k\in K_{ij}}$ to precompute and corresponding scalars $(\gamma^k_{ij})_{ij\in\{1,\dots,m\},k\in K_{ij}}$ that will be recomputed each time. The distinction between X and the γ^k_{ij} is given by the individual costs of each value. In order to keep the memory usage small, X should only contain a limited number of costly evaluations. Multiple appearances of the same value in X are to be understood as a single evaluation. The computational complexity to build a matrix M, can then be expressed as

$$\mathfrak{F}(build\ M) = \mathfrak{F}(precompute\ X) + \sum_{i,j\in\{1,\dots,m\},k\in K_{ij}} (\mathfrak{F}(\gamma^k_{ij}) + 1) \tag{70}$$

where $\mathfrak{F}(precompute\ X)$ and $\mathfrak{F}(\gamma^{(k)}_{ij})$ are the computational complexity respectively for the precomputation of the variables x^k_{ij} and the coefficients $\gamma^{(k)}_{ij}$ (including one flop for the multiplication $\gamma^{(k)}_{ij} x^k_{ij}$) plus one flop corresponding to summing the factor $M^{(k)}_{ij}$ into the matrix entry M_{ij}.

Additionally there can be factors $M^{(k)}_{ij}$ which appear multiple times, i.e. $\exists(i_1, j_1, k_1), (i_2, j_2, k_2)$ s.t. $M^{(k_1)}_{i_1 j_1} = M^{(k_2)}_{i_2 j_2}$. An obvious example from GMGPolar is the fact that the central update of the nine-point stencil is the sum of the left, right, bottom, and top updates, as seen in Equation (68). We can then write:

$$\mathfrak{F}(build\ M) = \mathfrak{F}(precompute\ X) + \sum_{i,j,k\in E} (\mathfrak{F}(\gamma^{(k)}_{ij}) + \sigma^k_{ij}) \tag{71}$$

where E defines the set of factors $K^{(k)}_{ij}$ such that factors $M^{(k)}_{ij}$ were factorized, and σ^k_{ij} is the number of occurrences of the factor $M^{(k)}_{ij}$, i.e., which can be reused $\sigma^k_{ij} - 1$ times. Finally, if there are factors which are numerically different but correspond to the same computational cost and number of occurences, we can write

$$\mathfrak{F}(build\ M) = \mathfrak{F}(precompute\ X) + \sum_{i,j,k\in\widetilde{E}} (\mathfrak{F}(\gamma^{(k)}_{ij}) + \sigma^k_{ij})\phi^{(k)}_{ij} \tag{72}$$

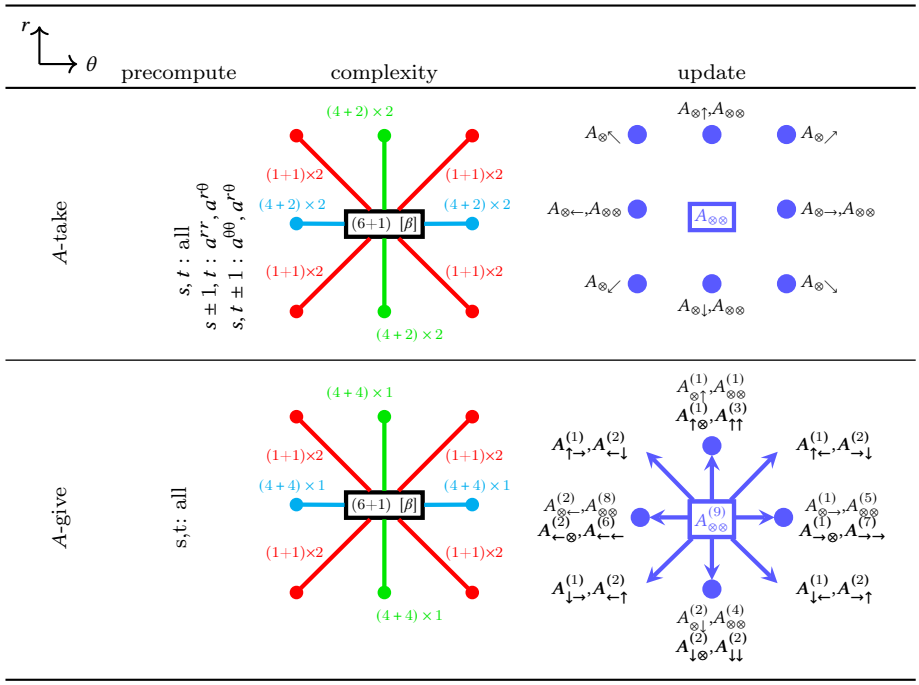
where $\widetilde{E}$ was reduced further to factorize factors with the same cost, and $\phi^{(k)}_{ij}$ gives the number of factors considered to have the same cost and number of occurences as $M^{(k)}_{ij}$ for the final calculus of the computational complexity. To simplify, setting aside the precomputation of the x^k_{ij} elements, for each aggregated factor in $\widetilde{E}$ the cost is decomposed as

$$(\#flops + \#occurences) \times \#factors. \tag{73}$$

Construction of A:

Upon implementation, one should take care of precomputing as much variables as possible, and computing only once factors which are used several times. How this can be done depends on the code. The general idea is to loop over all nodes and compute the entries of the matrix corresponding to each node independently. As discussed in Section 5.2.1, we can consider two approaches.

Table 6: Stencil representations of the construction of A in terms of computations for a single grid node (s,t). Two implementations are considered: A-take and A-give. Functions to precompute for the current node are given. The *complexity* column gives for each point in the stencil what is the computational cost expressed as **(#flops + #occurrences)×#factors**, where #flops is the complexity of computing the scaling coefficient of each factor, #occurrences is the number of times this factor appears in the stencil updates (= 1 addition in the matrix entry), and #factors is the number of different factors to compute with the same complexity. The colored lines indicate which function a^{xx} is required for each computation: green, blue, and red respectively for a^{rr}, $a^{\theta\theta}$, and $a^{r\theta}$. The *update* column gives the factors in the stencil corresponding to each complexity. The "$*$" symbols indicate which factors are computed for the stencil of a neighboring grid node, indicated by the arrow direction. Other factors are updated in the stencil of the current node. Purple dots show the updated stencil points, and a rectangle stands for the central update.



A-take: In a first approach, the loop is performed over each node (s,t) for which all entries in the corresponding row are computed. In this approach, we precompute all the functions a^{xx} required for the stencil, i.e. all three functions $(a^{rr}, a^{\theta\theta}, a^{r\theta})$ for the current node and its neighbors (top, left, bottom, right) except $a^{\theta\theta}$ for the top and bottom nodes, and a^{rr} for the left and right nodes. Based on Algo. 6, we get that $\mathfrak{F}_m(precompute) = 5\mathfrak{F}_m(DF) + 79$ flops.

Using the notation above, we then consider that each appearance of the functions a^{xx} or the β coefficient in the stencil (68) defines a factor, resulting in 25 factors. The factors are numbered in their order of appearance for each update in the stencil. It is easy for each of these factors to get the corresponding number of flops. In fact, all of the 8 diagonal factors require 1 flop each and appear only once, e.g. $A^{(1)}_{\otimes\nearrow} = a^{r\theta}_{\uparrow}/4$ appears only in $A_{\otimes\nearrow}$. The top, bottom, left, and right factors require 4 flops each and appear twice, e.g. $A^{(1)}_{\otimes\uparrow} = -\frac{(k_t+k_{t-1})}{2h_s} a^{rr}_{\otimes}$ appears in the top and central updates. Finally, there is a single factor corresponding to the β coefficient which requires 7 flops.

We give a representation of this approach in the first line of Table 6. We draw one diagram, called *complexity*, displaying the computational complexity corresponding to all factors, under the form **(#flops + #occurences) × #factors**. For example, on the top place of the stencil, we write the complexity $(4+2)\times 2$ corresponding to each factor $A^{(1)}_{\otimes\uparrow}$ and $A^{(2)}_{\otimes\uparrow}$, both needed in the top and central update. A second diagram, called *update*, specifies for each complexity, which factors of the stencil are computed. By summing all of these complexities, we then obtain the final complexity per node

$$\mathfrak{F}_m(build\ A) = \mathfrak{F}_m(precompute) + 71 = 5\mathfrak{F}_m(DF) + 150 \text{ flops}. \tag{74}$$

A-give: Alternatively, the A-give approach ensures that each function a^{xx} is only computed once. In this approach, the loop is performed over each node, where the local functions a^{xx} are computed. The corresponding complexity is $\mathfrak{F}_m(precompute) = \mathfrak{F}_m(DF)+19$. Then, all the factors requiring these a^{xx} functions are computed to update the matrix. These factors are part of the matrix entries in the row and column corresponding to the current node.

It is then possible, as before, to simply compute factors for the stencil of the current node, e.g. $A^{(1)}_{\otimes\uparrow} = -\frac{(k_t+k_{t-1})}{2h_s} a^{rr}_{\otimes}$ is involved in the top and central updates. However, this same factor also appears in the bottom and, again, in the central update of the neighboring node placed above in the grid, i.e. in $A_{\uparrow\otimes}$ and $A_{\uparrow\uparrow}$. The corresponding complexity for this factor is then $(4+4)\times 1$. Compared to the A-take approach, less factors are computed per node but their number of occurrence is increased.

We give again the representation of this approach in Table 6. In the *update* column, we additionally indicate with a "$*$" symbol which factors are computed to update the stencil of a neighboring grid node, with an arrow showing which neighbor. Factors without a "$*$" symbol simply correspond to updates for the current node. Note that the factors for the diagonal updates of the stencil require only the functions a^{xx} from the neighboring nodes. For example, the top-right update depends on the functions $a^{r\theta}$ of the top and right neighboring nodes. Thus, there never is a purple dot in the corresponding representation. By combinating all the factors, we then obtain the final complexity per node

$$\mathfrak{F}_m(build\ A) = \mathfrak{F}_m(precompute) + 55 = \mathfrak{F}_m(DF) + 74 \text{ flops}. \tag{75}$$

The computational cost for the construction of A is thus much lower when using the A-give approach. This is especially true for complex geometries such as Czarny (see Algo. 5), where computing the Jacobian of the mapping would become the dominating part when building the matrix.

In terms of memory, the matrix A contains nine entries per row and thus we have $\mathfrak{M}_m(A) = 9$. This cost only appears on the coarsest grid level where a direct solver is used to solve a linear system based on A. On the other levels, A is applied with a matrix-free approach.

Matrix-free application (with A-give approach):

As we advocate to use the favorable A-give approach, we only present the final result for A-give. The result for A-take can be computed easily. As detailed in Section 5.1, the computational complexity from applying a matrix in a matrix-free fashion can be decomposed in its construction cost $\mathfrak{F}_m(build\ A)$, and the cost for its application $\mathfrak{F}_m(apply\ A)$. The latter corresponds to the number of factors in the stencil. With (75), we thus have

$$\begin{aligned} \mathfrak{F}_m(apply\ A) &= 25, \\ \mathfrak{F}_m(matrix\text{-}free\ Au) &= \mathfrak{F}_m(build\ A) + \mathfrak{F}_m(apply\ A) = \mathfrak{F}_m(DF) + 99. \end{aligned} \tag{76}$$

8.4 Smoothing

8.4.1 $A_{s_c s_c}$ and $A_{s_c s_c^\perp}$ matrices

We now detail the complexity from the construction of the matrices $A_{s_c s_c}$ and the matrix-free application of the $A_{s_c s_c^\perp}$. For the same reasons as for the matrix A in section 8.3, we follow an A-give type of approach. We focus on the case without extrapolation first, then detail the differences when using extrapolation.

Smoothing without extrapolation:
When constructing the operators, either $A_{s_c s_c}$ or $A_{s_c s_c^\perp}$, we need to loop over all nodes connected to the nodes of the current partial smoother, i.e., the nodes with the same color (from s_c), and the nodes with the other color (from $s_c^\perp$). For example, when handling the C_W (circle white) partial smoother, we also need to loop over the nodes corresponding to the $C_W^\perp = C_B$ partial smoother since these nodes are linked through the nine point stencil. Also, in order to update the nodes of the outermost circle line of the circle partial smoother, we need to loop over the nodes in the first circle line of the radial partial smoother. This line is classically called a *halo*. This is explained in Section 5.3 in the main body of the article.

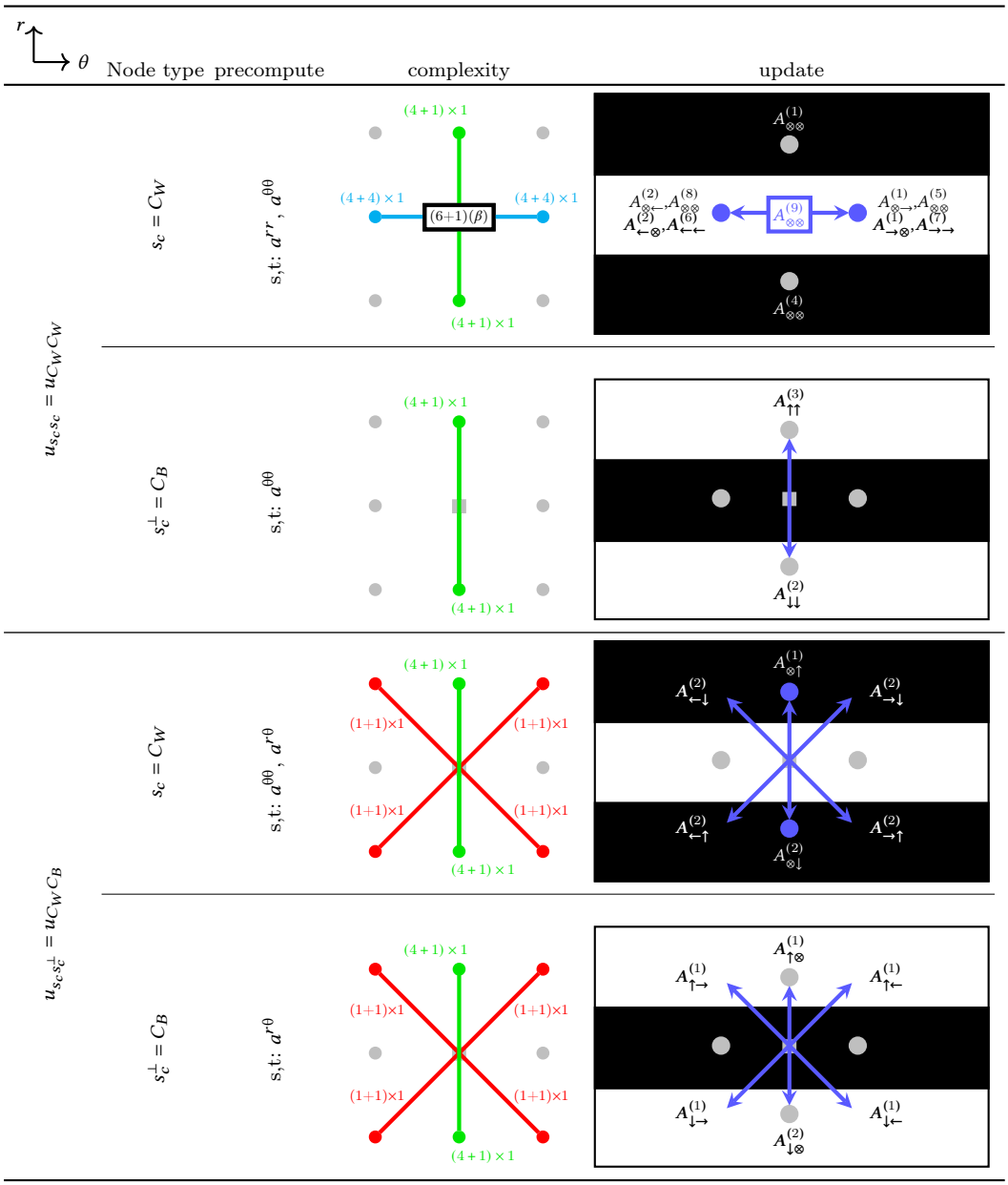
To get the corresponding computational complexities, we use the same approach and notation as for the matrix A. We thus detail the complexity stencils in Table 7 in the case of the circle white smoother (C_W), distinguishing when the current node is from the current partial smoother, or has the other color. The complete complexity for the smoother C_W is then obtained by summing these two cases.

Remark 6 Note that if we merge all the stencils in Table 7 ($A_{s_c s_c}$ and $A_{s_c s_c^\perp}$, for a node s_c and a node $s_c^\perp$), we reconstruct the original 9-point stencil for A, see Table 6.

The complexities per node for the radial partial smoother are obtained with a 90° rotation, with identical computational cost per node. Also, there is no difference in computational cost per node depending on which partial smoother is applied. Thus, all partial smoothers behave in the same way. Using Table 7 similarly as for the matrix A, we obtain the final complexities per node

$$\begin{aligned} \mathfrak{F}_m(build\ A_{s_c s_c}) &= (14 + \mathfrak{F}_m(DF) + 33) && [s_c] \\ &\quad + (10 + \mathfrak{F}_m(DF) + 10) && [s_c^\perp] \\ &= 2\mathfrak{F}_m(DF) + 67, \\ \mathfrak{F}_m(build\ A_{s_c s_c^\perp}) &= (15 + \mathfrak{F}_m(DF) + 18) + (11 + \mathfrak{F}_m(DF) + 18) \\ &= 2\mathfrak{F}_m(DF) + 62. \end{aligned} \tag{77}$$

Table 7: Stencil representations for the construction of $A_{s_c s_c}$ and $A_{s_c s_c^\perp}$ for the circle white partial smoother ($A_{C_W C_W}$, $A_{C_W C_B}$) in the case without extrapolation. This representation follows the same principles as Table 6. We distinguish two cases for each matrix: when the current node is of the same color as the partial smoother (white), or when it is of the other color (black). Updates of the local stencil are only present when considering a node of the same color as the partial smoother. Note that all other partial smoothers behave in the same way.



Finally, in order to apply $A_{s_c s_c^\perp}$ in a matrix-free fashion, as detailed in Section 5.1, we only need the additional application cost $\mathfrak{F}_m(apply\ A_{s_c s_c^\perp}) = 12$ which corresponds to the number of

factors to compute $A_{s_c s_c^\perp}$. The latter number can be obtained by counting the factors from both s_c and $s_c^\perp$ cases in Table 6. Finally, we get

$$\mathfrak{F}_m(\textit{matrix-free } A_{s_c s_c^\perp} u) = \mathfrak{F}_m(\textit{build } A_{s_c s_c^\perp}) + \mathfrak{F}_m(\textit{apply } A_{s_c s_c^\perp}) = 2\mathfrak{F}_m(DF) + 74. \tag{78}$$

Smoothing with implicit extrapolation:
When using the implicit extrapolation, the smoother is applied only on the fine nodes. Let's consider without loss of generality that the first circle line of the grid ($r = r_1$) is white. Let's also remind that the coarse grids are defined using standard coarsening as detailed in Section 4.1. Thus, starting from the first grid node (r_1, θ_1), the coarse nodes are chosen with a distance of 2 from each other. With respect to $A_{s_c s_c}$ and $A_{s_c s_c^\perp}$, nothing changes in the partial smoothers for black nodes when using extrapolation because they are all fine nodes. However, concerning the white nodes, we distinguish the two cases, whether the current node in the loop is a fine node or a coarse node.

We provide Table 8, again representing the complexity per node as well as the computed factors for the construction of $A_{s_c s_c}$ and $A_{s_c s_c^\perp}$ for white nodes of the circle partial smoother. This table does not include the case of black nodes, which can still be found in Table 7 (case without extrapolation). As before, the complexities for the radial smoother are identical to those of the circle smoother.

Let us remember that the ratio of nodes for each color is expressed as $\mu_c = m_c/m \approx 1/2$, where m is the total number of grid points, and m_c the number of grid points of color $c \in \{B, W\}$. Then, the ratio of white fine and coarse nodes are $\mu_W^{fine} = \mu_W^{coarse} = \mu_W/2$.

Remark 7 The number of black lines is identical to the number of white lines plus or minus 1. Asymptotically, the previous approximations can be considered exact.

The complexity per node for black nodes is unchanged compared to the one without extrapolation. Following the same principles as before, using Table 8, we obtain

$$\begin{aligned}
\mathfrak{F}_m^{ex}(\textit{build } A_{s_c s_c}) &= \left[\mu_W^{fine}(14 + \mathfrak{F}_m(DF) + 27) + \mu_W^{coarse}(10 + \mathfrak{F}_m(DF) + 10) + \mu_B(10 + \mathfrak{F}_m(DF) + 10)\right] && [W] \\
&\quad + [\mu_B(14 + \mathfrak{F}_m(DF) + 33) + \mu_W(10 + \mathfrak{F}_m(DF) + 10)] && [B] \\
&= 2\mathfrak{F}_m(DF) + 50.5\mu_W + 67\mu_B \\
&\approx 2\mathfrak{F}_m(DF) + 58.75 \\
\mathfrak{F}_m^{ex}(\textit{build } A_{s_c s_c^\perp}) &= \left[\mu_W^{fine}(15 + \mathfrak{F}_m(DF) + 18) + \mu_W^{coarse}(15 + \mathfrak{F}_m(DF) + 20) + \mu_B(11 + \mathfrak{F}_m(DF) + 18)\right] && [W] \\
&\quad + [\mu_B(15 + \mathfrak{F}_m(DF) + 18) + \mu_W(11 + \mathfrak{F}_m(DF) + 18)] && [B] \\
&= 2\mathfrak{F}_m(DF) + 63\mu_W + 62\mu_B \\
&\approx 2\mathfrak{F}_m(DF) + 62.5 \\
\mathfrak{F}_m^{ex}(\textit{apply } A_{s_c s_c^\perp})) &= (4\mu_W^{fine} + 6\mu_W^{coarse} + 6\mu_B) + (6\mu_B + 6\mu_W) \\
&= 11\mu_W + 12\mu_B \\
&\approx 11.5 \\
\mathfrak{F}_m^{ex}(\textit{matrix-free } A_{s_c s_c^\perp} u) &= \mathfrak{F}_m^{ex}(\textit{build } A_{s_c s_c^\perp}) + \mathfrak{F}_m^{ex}(\textit{apply } A_{s_c s_c^\perp})) = 2\mathfrak{F}_m(DF) + 74
\end{aligned} \tag{79}$$

Thus, when activating the extrapolation, the cost of applying $A_{s_c s_c^\perp}$ stays the same, and the cost from building $A_{s_c s_c}$ is decreased.

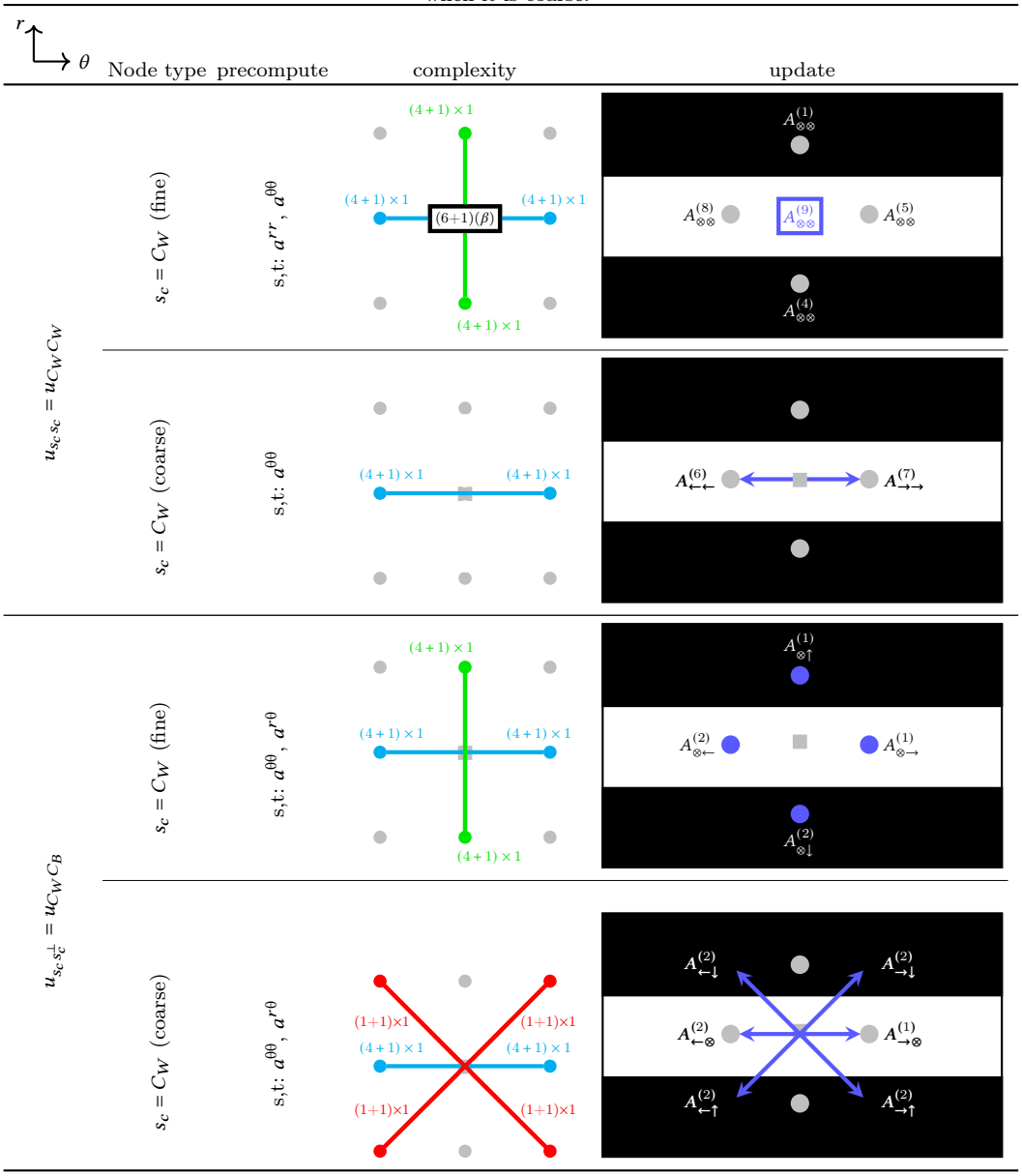
8.4.2 LU factorization for the partial smoothing

To apply the partial smoothers, it is necessary to solve the equation

$$A_{s_c s_c} u = z \tag{80}$$

Table 8: Stencil representations for the construction of $A_{s_c s_c}$ and $A_{s_c s_c^\perp}$ for white nodes when using the circle white partial smoother

$(A_{C_W C_W}, A_{C_W C_B})$, in the case where implicit extrapolation is used. This representation follows the same principles as Table 7. Additionally, we distinguish two cases for each matrix: when the current node is fine or when it is coarse.



which can be done with the combination of an LU factorization, using a modified Thomas' algorithm, and the corresponding backward/forward substitution. Starting from the particular sparsity pattern of $A^{(i)}_{s_c s_c}$ (see Section 5), we consider a separate line i of size n.

– Circle smoother: tridiagonal matrix with periodicity conditions on the first and last rows. Using Algo. 7, it is possible to compute an LU factorization of the matrix with $10n$ flops. Using this factorization, it is possible to solve Equation (80) with $9n$ flops using Algo. 8. Due to the last row and column filling-in, the resulting memory consumption is $\mathfrak{M}_m(\mathrm{LU}(A_{s_c s_c})) = 5$. In order to save some memory footprint, the LU decomposition is written directly inside the $A_{s_c s_c}$ matrix. L is in the lower triangular part of the modified $A^{(i)}_{s_c s_c}$ with an omitted diagonal of ones, U is the diagonal and upper triangular part of the modified $A^{(i)}_{s_c s_c}$.
– Radial smoother: tridiagonal matrix. Using Algo. 9, it is possible to compute an LU factorization of the matrix with $3n$ flops. Using this factorization, it is possible to solve Equation (80) with $5n$ flops using Algo. 10. There is no fill-in, so the resulting memory consumption is $\mathfrak{M}_m(\mathrm{LU}(A_{s_c s_c})) = \mathfrak{M}_m(A_{s_c s_c}) = 3$. Again, the LU decomposition is directly written over $A_{s_c s_c}$.

When using the extrapolation, the difference is that only the fine nodes are smoothed. Then, in the case of the fine nodes in the white smoother, the $A_{s_W s_W}$ matrix becomes diagonal and can be inverted trivially for 1 flop per entry. We remind the reader that the white nodes represent half of all nodes, i.e., asymptotically we have $\mu_{s_W} = \mu_{s_B}$ and $\mu_W = \mu_B = 1/2$, and that all nodes are either white or black so $\mu_C + \mu_R = 1$. By summing all of the lines, we finally get:

– without extrapolation:

$$\begin{aligned} \mathfrak{F}_m(\mathrm{LU}(A_{s_c s_c})) &= 10\mu_C + 3\mu_R = 3 + 7\mu_C \\ \mathfrak{F}_m(\textit{compute } A^{-1}_{s_c s_c} z) &= 9\mu_C + 5\mu_R = 5 + 4\mu_C \end{aligned} \tag{81}$$

– with extrapolation, since only the fine nodes are smoothed, i.e., half of the white colored nodes:

$$\begin{aligned} \mathfrak{F}_m(\mathrm{LU}(A_{s_c s_c})) &= 0\mu_W/2 + 10\mu_C/2 + 3\mu_R/2 = 1.5 + 3.5\mu_C \\ \mathfrak{F}_m(\textit{compute } A^{-1}_{s_c s_c} z) &= \mu_W/2 + 9\mu_C/2 + 5\mu_R/2 = 2.75 + 2\mu_C \end{aligned} \tag{82}$$

Algorithm 7 Compute $A^{(i)}_{C_c C_c} = L^{(i)}_{C_c C_c} U^{(i)}_{C_c C_c}$: $O(10n)$ (here n is the length of a circle line)

Input: $A^{(i)}_{C_c C_c}$
Output: $A^{(i)}_{C_c C_c}$ contains the LU decomposition
1: **for** $i = 1, \cdots, n-2$ **do**
2: // update row $i+1$
3: $A(i+1,i) = A(i+1,i)/A(i,i)$ `(1 div.)`
4: $c = A(i+1,i)$
5: $A(i+1,i+1) = A(i+1,i+1) - c\,A(i,i+1)$ `(1 add, 1 mult.)`
6: $A(i+1,n) = A(i+1,n) - c\,A(i,n)$ `(1 add, 1 mult.)`
7: // update last row
8: $A(n,i) = A(n,i)/A(i,i)$ `(1 div.)`
9: $c = A(n,i)$
10: $A(n,i+1) = A(n,i+1) - c\,A(i,i+1)$ `(1 add, 1 mult.)`
11: $A(n,n) = A(n,n) - c\,A(i,n)$ `(1 add, 1 mult.)`
12: **end for**

8.4.3 Complete smoothing

The only step left is to perform the difference between f_{s_c} and $A_{s_c s_c^\perp} u$ for $\mathfrak{F}_m(f_{s_c} - A_{s_c s_c^\perp} u)$ flop per node. Separating the setup, which is only performed once, from a relaxation step performed

Algorithm 8 Solve $L^{(i)}_{C_cC_c} U^{(i)}_{C_cC_c} x = z$: $\mathcal{O}(9n)$ (here n is the length of a circle line)

Input: $L^{(i)}_{C_cC_c} U^{(i)}_{C_cC_c}$ decomposition
Output: $x = (A^{(i)}_{C_cC_c})^{-1} z$
// Forward substitution: $Ly = z$
for $i = 1, \cdots, n-2$ **do**
  $y(i) = z(i)$
  $z(i+1) = z(i+1) - A(i+1,i)\, y(i)$ `(1 add, 1 mult.)`
  $z(n) = z(n) - A(n,i)\, y(i)$ `(1 add, 1 mult.)`
end for
$y(n-1) = z(n-1)$
$z(n) = z(n) - A(n,n-1)\, y(n-1)$ `(1 add, 1 mult.)`
$y(n) = z(n)$
// Backward substitution: $Ux = y$
$x(n) = y(n)/A(n,n)$ `(1 div.)`
$y(n-1) = y(n-1) - A(n-1,n)\, x(n)$ `(1 add, 1 mult.)`
for $i = n-1, \cdots, 2$ **do**
  $x(i) = y(i)/A(i,i)$ `(1 div.)`
  $y(i-1) = y(i-1) - A(i-1,i)\, x(i) - A(i-1,n)\, x(n)$ `(2 add, 2 mult.)`
end for
$x(1) = y(1)/A(1,1)$ `(1 div.)`

Algorithm 9 Compute $A^{(i)}_{R_cR_c} = L^{(i)}_{R_cR_c} U^{(i)}_{R_cR_c}$: $\mathcal{O}(3n)$ (here n is the length of a radial line)

Input: $A^{(i)}_{R_cR_c}$
Output: $A^{(i)}_{R_cR_c}$ contains the LU decomposition
for $i = 1, \cdots, n-1$ **do**
  $A(i+1,i) = A(i+1,i)/A(i,i)$ `(1 div.)`
  $A(i+1,i+1) = A(i+1,i+1) - A(i,i+1)\, A(i+1,i)$ `(1 add, 1 mult.)`
end for

Algorithm 10 Solve $L^{(i)}_{R_cR_c} U^{(i)}_{R_cR_c} x = z$: $\mathcal{O}(5n)$ (here n is the length of a radial line)

Input: $L^{(i)}_{R_cR_c} U^{(i)}_{R_cR_c}$ decomposition
Output: $x = (A^{(i)}_{R_cR_c})^{-1} z$
// Forward substitution: $Ly = z$
for $i = 1, \cdots, n-1$ **do**
  $y(i) = z(i)$
  $z(i+1) = z(i+1) - A(i+1,i)\, y(i)$ `(1 add, 1 mult.)`
end for
$y(n) = z(n)$
// Backward substitution: $Ux = y$
for $i = n, \cdots, 2$ **do**
  $x(i) = y(i)/A(i,i)$ `(1 div.)`
  $y(i-1) = y(i-1) - A(i-1,i)\, x(i)$ `(1 add, 1 mult.)`
end for
$x(1) = y(1)/A(1,1)$ `(1 div.)`

during the multigrid cycles, we obtain the final complexity as

$$
\begin{aligned}
\mathfrak{F}_m(\textit{smoother setup}) &= \mathfrak{F}_m(\textit{build } A_{s_cs_c}) + \mathfrak{F}_m(\mathrm{LU}(A_{s_cs_c})) \\
&= \left\{ \begin{array}{ll} (2\mathfrak{F}_m(DF) + 67) + (3 + 7\mu_C), & \text{without extrapolation} \\ (2\mathfrak{F}_m(DF) + 58.75) + (1.5 + 3.5\mu_C), & \text{with extrapolation} \end{array} \right\}, \\
&= \left\{ \begin{array}{ll} 2\mathfrak{F}_m(DF) + 70 + 7\mu_C, & \text{without extrapolation} \\ 2\mathfrak{F}_m(DF) + 60.25 + 3.5\mu_C, & \text{with extrapolation} \end{array} \right\}, \\
\mathfrak{F}_m(\textit{smoother apply}) &= \mathfrak{F}_m(\textit{matrix-free } A_{s_cs_c^\perp} u) + \mathfrak{F}_m(\textit{compute } A^{-1}_{s_cs_c} z) + \mathfrak{F}_m(f_{s_c} - A_{s_cs_c^\perp} u) \\
&= \left\{ \begin{array}{ll} (2\mathfrak{F}_m(DF) + 74) + (5 + 4\mu_C) + 1, & \text{without extrapolation} \\ (2\mathfrak{F}_m(DF) + 74) + (2.75 + 2\mu_C) + (1 - \mu_W/2), & \text{with extrapolation} \end{array} \right\}, \\
&= \left\{ \begin{array}{ll} 2\mathfrak{F}_m(DF) + 80 + 4\mu_C, & \text{without extrapolation} \\ 2\mathfrak{F}_m(DF) + 77.5 + 2\mu_C, & \text{with extrapolation} \end{array} \right\}, \\
\mathfrak{M}_m(\textit{smoother setup}) &= \left\{ \begin{array}{ll} 5\mu_C + 3\mu_R = 3 + 2\mu_C, & \text{without extrapolation} \\ \mu_W/2 + 5\mu_C/2 + 3\mu_R/2 = 1.75 + \mu_C, & \text{with extrapolation} \end{array} \right\}.
\end{aligned}
\tag{83}
$$