Bergische Universität Wuppertal

Fakultät für Mathematik und Naturwissenschaften

Institute of Mathematical Modelling, Analysis and Computational Mathematics (IMACM)

Preprint BUW-IMACM 20/53

Jens Hahne, Stephanie Friedhoff, and Matthias Bolten

PyMGRIT: A Python Package for the parallel-in-time method MGRIT

November 11, 2020

http://www.imacm.uni-wuppertal.de

Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

PyMGRIT: A Python Package for the parallel-in-time method MGRIT

Jens Hahne* | Stephanie Friedhoff | Matthias Bolten

[1]Department of Mathematics, Bergische Universität Wuppertal, 42097 Wuppertal, Germany, Email: {jens.hahne, friedhoff, bolten}@math.uni-wuppertal.de

Correspondence
*Email: jens.hahne@math.uni-wuppertal.de

Summary

In this paper, we introduce the Python framework `PyMGRIT`, which implements the multigrid-reduction-in-time (MGRIT) algorithm for solving the (non-)linear systems arising from the discretization of time-dependent problems. The MGRIT algorithm is a reduction-based iterative method that allows parallel-in-time simulations, i. e., calculating multiple time steps simultaneously in a simulation, by using a time-grid hierarchy. The `PyMGRIT` framework features many different variants of the MGRIT algorithm, ranging from different multigrid cycle types and relaxation schemes, as well as various coarsening strategies, including time-only and space-time coarsening, to using different time integrators on different levels in the multigrid hierachy. The comprehensive documentation with tutorials and many examples, the fully documented code, and a large number of pre-implemented problems allow an easy start into the work with the package. The functionality of the code is ensured by automated serial and parallel tests using continuous integration. `PyMGRIT` allows serial runs for prototyping and testing of new approaches, as well as parallel runs using the Message Passing Interface (MPI). Here, we describe the implementation of the MGRIT algorithm in `PyMGRIT` and present the usage from both user and developer point of views. Three examples illustrate different aspects of the package, including pure time parallelism as well as space-time parallelism by coupling `PyMGRIT` with `PETSc` or `Firedrake`, which enable spatial parallelism through MPI.

KEYWORDS:
Multigrid-reduction-in-time (MGRIT), parallel-in-time integration

1 | INTRODUCTION

The classical approach for solving an initial value problem is based on a time-stepping procedure, which computes the solution at a point in time based on the solution at one or more previous time steps and, thus, propagates the solution over time. The method is optimal, i. e., of order $\mathcal{O}\left(N_t\right)$ for N_t time steps, and gives the solution after N_t applications of the time integrator. However, time-stepping algorithms are completely sequential in time and limit potential parallelization to the spatial domain. In recent years, the structure of modern computer systems has been characterized by a growing number of processors, as the clock rates of the individual processors stagnate. As a consequence, possibilities of spatial parallelization are quickly exhausted, although further resources are available. Promising approaches for using these modern computer systems to further reduce the runtime of simulations of initial value problems are parallel-in-time methods, which allow not only spatial but also temporal parallelism.

One of these approaches is the iterative multigrid-reduction-in-time (MGRIT) algorithm[1], which applies multigrid reduction principles to the time domain. The idea of the algorithm is based on using a hierarchy of time grids, where the finest grid contains the same points in time as in the time-stepping approach and the number of points on the coarser grids decreases, so that the coarsest grid consists of only a few points. While time integration is applied to the coarsest grid sequentially, time subdomains are handled in parallel on the other grids. Over the recent years, the algorithm has been successfully applied to various problems, e. g., linear and nonlinear parabolic problems[1,2], compressible fluid dynamics[3], power grids[4,5], eddy-current simulations[6–8], linear advection[9,10], and machine learning[11].

Besides the MGRIT algorithm, there are many other iterative and direct parallel-in-time methods. Probably the most well-known method is Parareal[12], whose success is due to its simplicity and applicability: For the use of the algorithm only an expensive and a cheap time integrator is needed. This idea of Parareal can be interpreted in a variety of frameworks of numerical schemes. In particular, Parareal can be seen as a two-level MGRIT variant[1]. The parallel full aproximation scheme in space and time (PFASST)[13] is based on spectral deferred corrections (SDC)[14] and allows space-time parallelism by simultaneously executing several SDC "sweeps" on a space-time hierarchy. Another method, the revisionist integral deferred correction method (RIDC)[15,16] allows time parallelism through the pipelined parallel application of integral deferred corrections. A more detailed, structured, and general overview of parallel-time-integration approaches is given in[17]. Furthermore, the website `https://parallel-in-time.org` offers many more references and information about parallel-in-time methods.

Despite the increasing number of algorithms, concepts, and papers in the field of parallel-in-time integration, the number of accessible stand-alone parallel-in-time libraries providing parallelization in time or allowing space-time parallelism is relatively small. The following libraries are most noteable: The XBraid[18] library, written in C provides an implementation of the MGRIT algorithm and libridc[16] is a C++ implementation of the RIDC algorithm. The PFASST method is implemented by various libraries which are summarized in[19], in particular by the Fortran library `libpfasst`, `PFASST++` written in C++, and the Python implementation `pySDC`. Besides these, there are a number of other implementations of parallel-in-time concepts, but most of them are either more specialized or more difficult to access, such as the `SWEET` code[20] or various implementations of the PARAEXP method[21]. This paper describes the new Python framework `PyMGRIT`, which implements the MGRIT algorithm. `PyMGRIT` is designed to be easy to use, ranging from the very simple installation of the package to calling or overriding functions, making it ideal for testing the application of MGRIT to problems, or for prototyping new ideas and strategies for the MGRIT algorithm. More precisely, `PyMGRIT` allows on the one hand the simple, pure application of different variations of the MGRIT algorithm to problems without having to worry about implementation details, parallel communication, etc., but on the other hand, it allows flexible adaptations of individual components of the algorithm by overriding existing functions. This allows users to try MGRIT for their application, and it makes `PyMGRIT` particularly attractive for the training of students. Furthermore, `PyMGRIT` allows easy coupling with other software that already exists as Python packages, e. g., `Matplotlib`[22] for creating static and interactive visualizations or `Firedrake`[23] for using the Finite Element Method (FEM). Additionally, `PyMGRIT` allows space-time parallel runs that go far beyond prototyping.

The following sections of this paper are organized as follows: Section 2 describes the MGRIT algorithm. Based on this, Section 3 introduces the `PyMGRIT` framework and describes the implementation of the algorithm and its variations. Then, the use of `PyMGRIT` is examined, describing first basic usage of the package, followed by a description of implementing a custom application using `PyMGRIT`. In Section 4, different aspects and possibilities of using `PyMGRIT` are demonstrated in three numerical examples. Finally, conclusions are drawn and possible extensions of the package are discussed in Section 5.

2 | MGRIT

The idea of MGRIT is to enable parallelism in a traditionally sequential process. For time-dependent problems, starting with an initial value, the solution is classically computed step by step, i. e., starting with the initial condition, the solution at each time step is computed based on the solution at one or more previous time steps. This sequential time-stepping process is shown in Figure 1, which displays the initial condition of a simple linear heat conduction problem along with the solutions after 512 and 1024 time steps.

In contrast, the iterative MGRIT algorithm solves the problem by updating the solution at many points in time simultaneously. Thereby, the initial guess of the solution can be chosen arbitrarily for all points in time. The idea of the algorithm is based on a multilevel hierarchy for the problem. While the finest level contains the same points in time as in the time-stepping approach, on coarse levels only a subset of these points are considered. Optimally, the coarsest grid contains only a very small number of

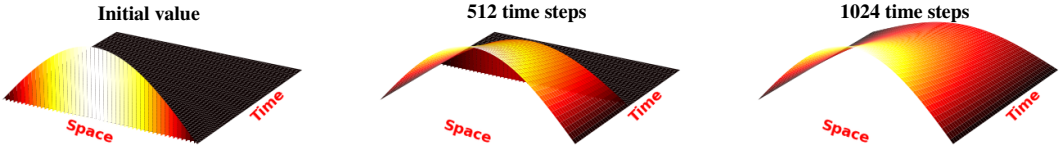

FIGURE 1 Sequential time stepping for a simple heat conduction problem. Starting with the initial condition (left), the solution is propagated over time (middle, right).

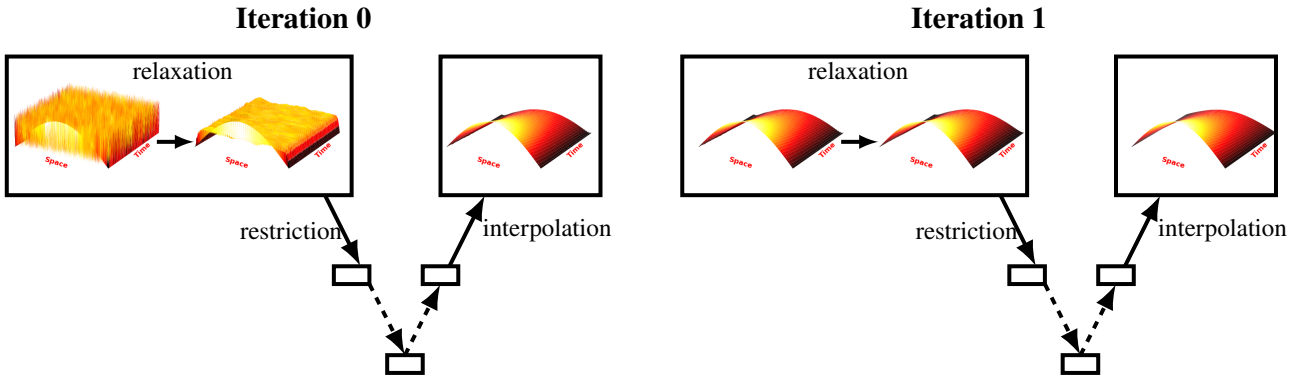

FIGURE 2 Two MGRIT iterations for a simple heat conduction problem.

points in time, e. g., three or five, and can therefore be solved exactly and quickly by the classical time-marching approach. This solution on the coarsest grid is used in the algorithm to update the solution on the finer grids until the solution on the finest grid is accurate enough based on a predefined tolerance. More precisely, one iteration of the algorithm consists of the following steps:

1. Relaxation on the finest grid for multiple points in time simultaneously. Relaxation corresponds to a local application of the time integrator, which is also used for time stepping.
2. Transferring the solution to the next coarser grid, which contains only a subset of the points in time.
3. Repeating steps 1 and 2 for the next coarser grid until the coarsest grid is reached.
4. Solving the problem on the coarsest grid directly.
5. Interpolating the solution from the coarsest grid to the finest grid, improving the solution on all grid levels.

These steps are applied iteratively until a desired quality of the solution is achieved. Figure 2 shows this process for two MGRIT iterations. While this is only a rough description of the algorithm, it reveals one of the key features of MGRIT: its non-intrusiveness. The only requirement for the algorithm is a time-stepping procedure, which is also used in the time-marching approach and, thus, this procedure already exists for many problems and can be integrated easily into a parallel framework with MGRIT.

After the previous schematic representation of time stepping and the MGRIT algorithm, we now want to examine both approaches algorithmically. Therefore, we consider an initial value problem of the form

$$\mathbf{u}'(t) = \mathbf{f}(t, \mathbf{u}(t)), \quad \mathbf{u}(t_0) = \mathbf{g}_0, \quad t \in (t_0, t_f], \tag{1}$$

which can, for example, be a system of ordinary differential equations after the spatial discretization of a space-time partial differential equation. We discretize the time interval on a grid with uniformly distributed points $t_i = i\Delta t,\ i = 0, 1, \ldots, N_t$ with

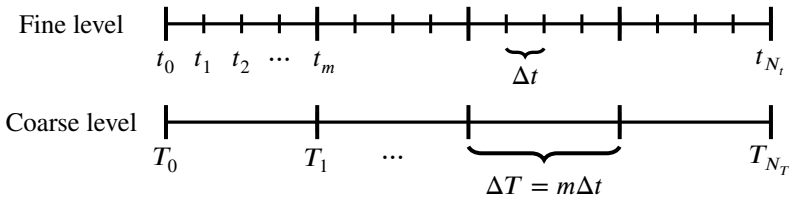

FIGURE 3 Fine and coarse temporal grid with uniformly distributed points. The coarse grid is created by removing all F-points (represented by short markers) of the fine grid.

constant time step $\Delta t = (t_f - t_0)/N_t$ and let $\mathbf{u}_i \approx \mathbf{u}(t_i)$ for $i = 0, \dots, N_t$ with $\mathbf{u}_0 = \mathbf{u}(0)$. Note that non-uniform distributions are also possible, but are not considered here for reasons of simplicity. Further, let $\mathbf{\Phi}_i$ be a time integrator, propagating the solution $\mathbf{u}_{i-1}$ from a time point t_{i-1} to time point t_i, including forcing from the right-hand side. Then, the one-step time integration method for the time-discrete initial value problem (1) can be written as

$$\mathbf{u}_i = \mathbf{\Phi}_i(\mathbf{u}_{i-1}), \quad i = 1, 2, \dots, N_t,$$

or, considering all time points at once, as the space-time system

$$\mathcal{A}(\mathbf{u}) \equiv \begin{bmatrix} \mathbf{u}_0 \\ \mathbf{u}_1 - \mathbf{\Phi}_1(u_0) \\ \vdots \\ \mathbf{u}_{N_t} - \mathbf{\Phi}_{N_t}(u_{N_{t-1}}) \end{bmatrix} = \begin{bmatrix} \mathbf{g}_0 \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \end{bmatrix} \equiv \mathbf{g}. \tag{2}$$

Sequential time-stepping solves problem (2) through a sequential forward solve, yielding the discrete solution after N_t applications of the time integrator and, thus, sequential time-stepping is optimal, i. e., of order $\mathcal{O}\left(N_t\right)$.

In order to embed the time integrator $\mathbf{\Phi}_i$ into a multi-level algorithm that iteratively solves problem (2), we first define the components of the algorithm and then construct the algorithm. More precisely, we define a coarse grid system, a restriction and a prolongation operator between temporal grids, and a relaxation scheme. For a given fine temporal grid $t_i = i\Delta t,\ i = 0, 1, \dots, N_t$, and a given integer coarsening factor $m > 1$, we define a splitting of the fine grid points into F- and C-points, where every m-th point is a C-point and all other points are F-points. Looking only at the C-points, we obtain a coarser grid $T_{i_c} = i_c \Delta T,\ i_c = 0, 1, \dots, N_T$, with $N_T = N_t/m$ and time step $\Delta T = m\Delta t$, as shown in Figure 3.

As the time-step size is different on the coarse grid, a time integrator $\mathbf{\Phi}_{i_c}$ is needed for the coarse level. We choose a re-discretization of the problem with time step ΔT, but several choices are possible, such as coarsening in the order of the discretization[24, 25]. Next, we define two types of relaxation strategies, i. e., schemes of local applications of the time integrator. The first scheme, the so-called F-relaxation, performs a relaxation of all F-points by propagating the solution from a C-point to all following F-points up to the next C-point. Each interval of F-points can be executed in parallel, whereby each interval consists of $m - 1$ sequential applications of the time integrator. The second scheme, the C-relaxation, analogously performs a relaxation of all C-points consisting of propagating the solution from the preceding F-point to a C-point. Again, all intervals of C-points can be updated simultaneously. Both relaxation schemes are shown in Figure 4 and can be combined to new schemes. The FCF-relaxation, i. e., an F-relaxation followed by a C- and another F-relaxation, is the typical choice for the MGRIT algorithm, but other combinations are also possible. Finally, we define two operators for the transfer between the temporal fine and coarse grid, more precisely a restriction and a prolongation operator. While restriction is an injection, we define the "ideal" prolongation as an injection at C-points followed by an F-relaxation.

Using the full approximation storage (FAS) framework[26] for solving both linear and nonlinear problems, the two-level MGRIT-FAS algorithm[3] extended by spatial coarsening[2] can be written as in Algorithm 1.

In algorithm 1, $\mathcal{A}_l(\mathbf{u}^{(l)}) = \mathbf{g}^{(l)}$ specifies the space-time problem on levels $l = 1, 2$, which can be either linear or nonlinear, and the two operators $\mathbf{R}_I$ and $\mathbf{P}$ denote the transfer operators between temporal grids. Additionally, the algorithm allows for spatial coarsening and we define $\mathbf{R}_s$ as spatial restriction and $\mathbf{P}_s$ as spatial prolongation. The relaxation scheme can be controlled by the parameter ν, whereby at least one F-relaxation is performed per iteration and then ν subsequent CF-relaxations are executed. Typically $\nu = 1$, i. e., an FCF-relaxation scheme, is chosen for the MGRIT algorithm. Note that the two-level variant with F-relaxation is equivalent to the parareal method[12]. To extend the two-level MGRIT-FAS algorithm to a multilevel setting, the

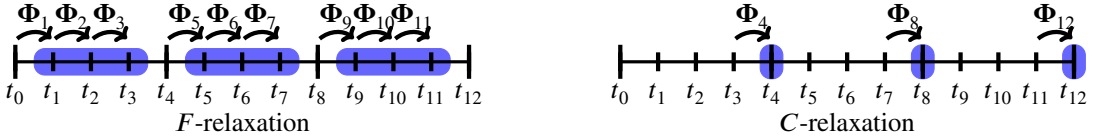

FIGURE 4 F- and C-relaxation for a temporal grid with 13 time points and a coarsening factor of four. In both relaxations, the independent intervals (blue) can be updated simultaneously.

Algorithm 1 MGRIT-FAS($\mathcal{A}, \mathbf{u}, \mathbf{g}$)

1: Apply F-relaxation to $\mathcal{A}_1(\mathbf{u}^{(1)}) = \mathbf{g}^{(1)}$
2: For 0 to ν:
3: Apply CF-relaxation to $\mathcal{A}_1(\mathbf{u}^{(1)}) = \mathbf{g}^{(1)}$
4: Inject the approximation and its residual to the coarse grid:
$\mathbf{u}^{(2)} = \mathbf{R}_I(\mathbf{u}^{(1)})$,
$\mathbf{g}^{(2)} = \mathbf{R}_I(\mathbf{g}^{(1)} - \mathcal{A}_1\mathbf{u}^{(1)})$
5: If spatial coarsening:
$\mathbf{u}^{(2)} = \mathbf{R}_s(\mathbf{u}^{(2)})$
$\mathbf{g}^{(2)} = \mathbf{R}_s(\mathbf{g}^{(2)})$
6: Solve $\mathcal{A}_2(\mathbf{v}^{(2)}) = \mathcal{A}_2(\mathbf{u}^{(2)}) + \mathbf{g}^{(2)}$
7: Compute the error approximation: $\mathbf{e} = \mathbf{v}^{(2)} - \mathbf{u}^{(2)}$
8: If spatial coarsening:
$\mathbf{e} = \mathbf{P}_s(e)$
9: Correct using ideal interpolation: $\mathbf{u}^{(1)} = \mathbf{u}^{(1)} + \mathbf{P}(\mathbf{e})$

two-level algorithm can be called resursively in step 6. Depending on the type and number of recursive calls, different multi-level variants can be defined.

The two-level MGRIT-FAS algorithm is based on an initial guess. This initial guess can be chosen arbitrarily; however, a good initial guess offers natural advantages for the runtime convergence of the algorithm. If prior knowledge of the solution exists, it should be chosen as an initial guess. If nothing is known, an improved initial guess can be computed by the nested iteration[27, 28] strategy. The idea of nested iteration is to compute an initial approximation on the coarsest level and to interpolate it to the finer levels. Note that, independent of the initial guess, the MGRIT algorithm provides the exact discrete solution of the fine grid after N_t/m or $N_t/(2m)$ iterations for F- or FCF-relaxation, respectively[1].

3 | THE PYMGRIT FRAMEWORK

In the previous section, we have familiarized ourselves with the way MGRIT works and how the algorithm looks like. In this section, we introduce the `PyMGRIT` framework. First, we describe the structure and availability of the framework. We then present the implementation of the MGRIT algorithm in `PyMGRIT`. In terms of algorithmic parameters, many choices must be made such as the relaxation scheme or the cycling strategy that lead to many different variants of the algorithm. `PyMGRIT` pursues two goals: On the one hand, `PyMGRIT` should enable the use of MGRIT with a minimal choice of algorithmic parameters. On the other hand, `PyMGRIT` should be as flexible as possible and give users the possibility to adjust all settings with little effort. Section 3.2 gives an overview of the most important MGRIT parameters and their implementation in `PyMGRIT`. Pursuing the first goal, most parameters have default values, but various choices are possible for pursuing the second goal. Section 3.3 presents a simple but typical code example for an application, while in Section 3.4, we take a developer's view of `PyMGRIT` and describe the classes required for implementing a custom application.

```
1 from pymgrit import Mgrit
2
3 mgrit = Mgrit(problem=problem)
4 mgrit.solve()
```

Listing 1: Example for the minimal generation of an instance of the class Mgrit. Only a problem hierarchy is required, all other parameters have default values.

3.1 | Availability and structure

The PyMGRIT framework consists of three components: the code repository[29], the documentation[30], and the Python Package Index (PyPI) version[31]. The installation of the package is typically done using `pip`, and only a few requirements have to be fulfilled. First, a Python version $\geq$ 3.6 is required and second, the following packages must be installed: `NumPy`[32], `SciPy`[33], `Matplotlib`[22], and `mpi4py`[34]. While `NumPy` and `SciPy` are used throughout the complete package, `Matplotlib` is mainly used for the visualization in different examples. The package `mpi4py` provides Python bindings for parallelization using the Message Passing Interface (MPI) standard. The requirements are listed in the file `setup.py` and will be installed automatically if PyPI and `pip` are used for the installation of the PyMGRIT package.
The code repository on GitHub[29] contains the latest version of the current code. After each commit on the master branch, `GitHub Actions` are used as a continuous integration tool to trigger automated processes. Automated serial and parallel tests are performed for all Python versions $\geq$ 3.6 using `pytest`[35] and `tox`[36]. Additionally, the code coverage of the tests is calculated and the coverage is automatically uploaded to Codecov[37], where the results can be viewed. Furthermore, an up-to-date version of the documentation is created using Sphinx. The documentation consists of two parts, a pure application programming interface (API) documentation and a documentation of the features of PyMGRIT. While the API documentation is automatically generated from the Python comments, the feature documentation includes a quick start, tutorial, and many examples for both basic and advanced usage of PyMGRIT. By creating a new release on GitHub, the current repository version is automatically uploaded to PyPI and is available via `pip` afterwards.
The PyMGRIT package primarily consists of four directories:

- src: this directory contains the core features of PyMGRIT, namely implementations of the MGRIT algorithm and of template classes of other required components. It also contains a number of sample problems and their implementation.
- examples: in this directory, a number of application examples can be found, both for basic and advanced usage of PyMGRIT, including all examples from the tutorial and from the documentation.
- tests: the test directory contains all written tests that are automatically executed after each commit on the master branch. Tests include core functions of PyMGRIT as well as sample applications.
- docs: contains the documentation for the examples, quick start, tutorial, etc.

3.2 | MGRIT in PyMGRIT

PyMGRIT is based on classes, with the exception of some auxiliary functions. Two steps are required to use the MGRIT algorithm: First, an instance of the `Mgrit` class from PyMGRIT's core must be created. In this step, all algorithmic choices and settings can be selected by parameters. The MGRIT implementation of PyMGRIT offers high flexibility for different variants of the algorithm, but at the same time the possibility to solve a problem with minimal specifications by using default values for most parameters. The second step is the actual solving of the problem, which is executed by simply calling a method of the `Mgrit` class.
Listing 1 provides a minimal example of creating an instance of the class with subsequent solving of the problem. Only the problem hierachy has to be passed to the class via constructor parameters in line 3 (the structure of a problem hierarchy is described in section 3.2.1). Other parameters of the constructor have default values, but can also be set manually. In line 4, the problem is then solved by calling the member function `solve`. The most important parameters of PyMGRIT's core class `Mgrit` and their effects are presented below; a complete list of all parameters can be found in the documentation.

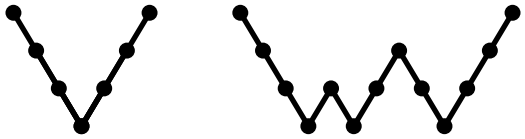

FIGURE 5 Structure of V- and F-cycles for four grid levels.

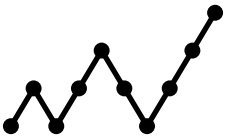

FIGURE 6 Structure of nested iteration when using four grid levels.

3.2.1 | Problem hierarchy and coarsening

The only mandatory parameter for instantiating an object of the `Mgrit` class is a problem hierarchy. This problem hierarchy is a list of problems (objects of classes that inherit from `PyMGRIT`'s core `Application` class) that defines the time-multigrid hierarchy. The exact structure of the problems and how to implement a problem in an application class is described in Section 3.4. At this point, it is sufficient to know that the time-multigrid hierarchy, i. e., the number of levels and the coarsening factor for the MGRIT solver, is given by this problem hierarchy. The problem for each level contains a vector with all time points for that level. The only requirement for the time points of each coarse level is that the time points are a subset of the time points of the finest level. This allows great flexibility in the choice of the coarsening factor, as it allows the implementation of regular coarsening strategies, coarsening strategies with different coarsening factors per level, and also semi-coarsening strategies very easily.

3.2.2 | Cycling and relaxation strategies

Two of the most important parameters for MGRIT are the cycling and the relaxation strategy. Both parameters have a significant impact on the convergence, parallelism, and efficiency of the algorithm. `PyMGRIT` provides two cycling strategies: V- and F-cycles. These can be controlled by the parameter `cycle_type`, i. e., by adding the cycling strategy via `cycle_type='V'` or `cycle_type='F'` to the constructor call. The structure of the two cycles is shown in Figure 5, which differs primarily in how often coarse grids are used. The V-cycle is the recursive implementation of algorithm 1, visiting the coarsest grid only once per MGRIT iteration. In contrast, the F-cycle visits the coarsest level several times per iteration, going back to the coarsest grid after having reached each level for the first time. As a consequence, an F-cycle better approximates the two-level method and, thus, faster convergence can be expected for F-cycles. However, F-cycles require more communication on intermediate coarse grids and more serial work on the coarsest grid, leading to worse parallel scalability compared to V-cycles for large numbers of processes.

Furthermore, an improved initial guess can be computed by using the nested iteration strategy, which can be controlled by the `Mgrit` constructor parameter `nested_iteration`. By default, nested iteration is active, but it can be disabled by passing `nested_iteration=False` to the constructor. As described at the end of Section 2, nested iteration computes an initial solution guess on the fine grid from coarse grids. More precisely, starting on the coarsest grid, an approximate solution is computed and interpolated to the next finer grid until the finest grid is reached. On each grid level, one V-cycle is used to compute an approximate solution, except on the coarsest grid, where the problem is solved directly. The resulting structure is illustrated in Figure 6. Note: In `PyMGRIT`, nested iteration is part of the setup phase of the algorithm, since it provides an initial solution guess on the finest grid.

The relaxation scheme in `PyMGRIT` can be set by the parameter `cf_iter`. This parameter specifies how many CF-relaxations follow an always executed F-relaxation. By default, the parameter is set to 1, i. e., an FCF-relaxation scheme. By varying this parameter, the user can easily switch between F-, FCF-, $FCFCF$-relaxation, and other relaxation schemes. Again, the choice of the scheme has a direct impact on the convergence of the algorithm. Usually, the more relaxation steps are performed, the

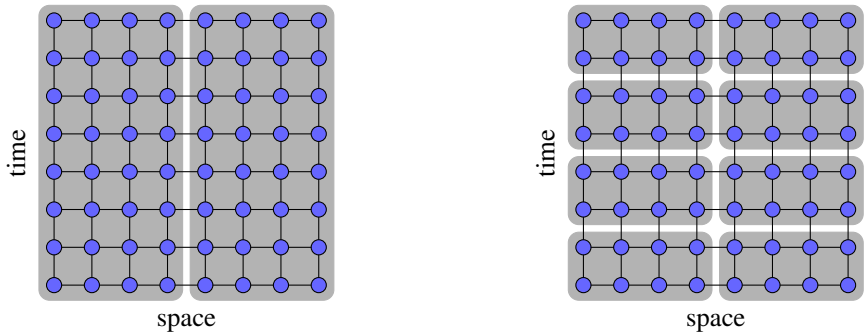

FIGURE 7 Decomposition of a space-time domain used for parallelizing computations in classical time-stepping algorithms (left) and in MGRIT (right).

better is the convergence. However, this improved convergence comes from the additional work performed in each iteration. The choice of the relaxation scheme also depends on the cycling strategy used. While an FCF-relaxation scheme is often a good starting point for a V-cycle, only F-relaxation is often sufficient for an F-cycle [1]. Note that the always performed F-relaxation on the finest grid is skipped after the second MGRIT iteration, because the updates during this F-relaxation are already performed during the post-relaxation of the ideal interpolation of the previous iteration.

3.2.3 | Stopping criterion

Another important parameter is the stopping tolerance of the algorithm, i. e., a threshold which stops the iterations if crossed. By default, `PyMGRIT` uses the absolute (space-)time residual of system (2) to check convergence. More precisely, denoting the residual of system (2) at the i_c-th C-point of the finest grid at iteration k by $r_{i_c}^{(k)}$, where i_c is an integer index for looping over all C-points, convergence is measured in the 2-norm based on the absolute (space-)time residual,

$$\|r^{(k)}\| = \left(\sum_{i_c} \|r_{i_c}^{(k)}\|^2 \right)^{1/2},$$

where the computation of the norm of the residual at each C-point, $\|r_{i_c}^{(k)}\|$, is specified by the user in a class that inherits from `PyMGRIT`'s core `Vector` class (see Section 3.4 for details). The convergence tolerance defines when to stop the iterations by checking $\|r^{(k)}\| < \texttt{tol}$ at each iteration, where the tolerance can be set using the constructor parameter `tol`; by default, `tol` is set to 10^{-7}. Currently, the only stopping criterion implemented in `PyMGRIT` is based on the (space-)time residual; however, the documentation includes examples of how to customize and change this criterion.

3.2.4 | Parallel decomposition and computation

In implementations based on a classical time-stepping approach, typically solution values are stored for one time point (or for a few time points in case of a multistep method) at a time, and values are updated at each step of the method. Therefore, such algorithms only offer the possibility to decompose the computational domain of the problem in space and, thus, allow spatial parallelism only to speed up the calculation. On the contrary, `PyMGRIT` stores the solution at every point in time and, thus, `PyMGRIT` provides another dimension for the decomposition of the computational domain: the time dimension. Figure 7 shows both decompositions of a space-time domain, reflecting the parallelization strategies applied in classical space-parallel time-stepping algorithms and in space-time-parallel MGRIT. Notice, that the partitioning of the temporal domain into time slices is added to the spatial decomposition, allowing a distribution of the time slices across additional parallel processing units. `PyMGRIT` provides a function `split_communicator` that splits a communicator into spatial and temporal components and returns two communicators for these components. The two communicators can be passed to the `Mgrit` constructor using the parameters `comm_x` for the space communicator and `comm_t` for the time communicator.

`PyMGRIT` distributes the time points of the finest grid equally over all processes contained in the time communicator. The distributions on coarser grids are based on the decompositions on their parent fine grids as each coarser grid contains a subset of their parent fine grid. The only exception is the coarsest grid, on which the problem is solved sequentially by one process.

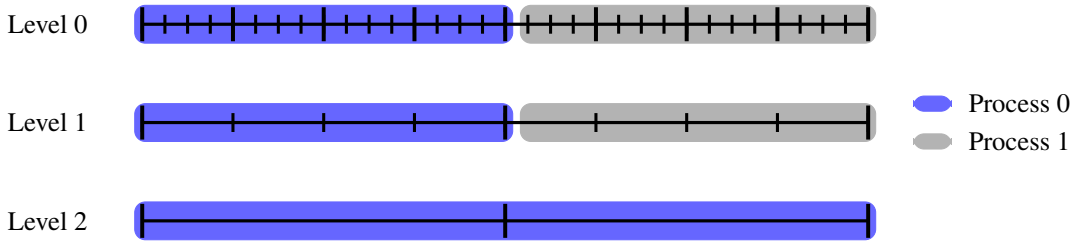

FIGURE 8 Example of the distribution of time points across two processes in `PyMGRIT`. The time points of the finest level are distributed evenly, and the coarser levels contain all C-points of their parent fine grids. On the coarsest grid, one process holds all time points.

This process then distributes the solution values to other processes. Figure 8 shows the distribution of $N_t = 33$ time points for two processes in a three-level setting with a coarsening factor of $m = 4$.
Storing all time points in the MGRIT algorithm enables (additional) parallelism, but at the same time increases memory requirements. For each time point, `PyMGRIT` creates an object for the solution at this time point. Additionally, on coarser levels, two additional objects must be stored for each time point: one for a restricted copy of the fine-grid object and one for the FAS right-hand side. There are several ways to reduce the increased memory requirements for the MGRIT algorithm, e. g., storing only C-points. Yet, this is not implemented in `PyMGRIT`, but it is a possible extension for future versions.

3.2.5 | Spatial coarsening

By default, `PyMGRIT` applies only temporal coarsening and does not require any additional information about the restriction or interpolation between spatial grids. If spatial coarsening is desired, the user can pass a list of spatial transfer operators using the `transfer` parameter. Thereby, a transfer operator is expected for each transfer between MGRIT levels, which ensures high flexibility for the user.

3.2.6 | Plots

The `PyMGRIT` class `MgritWithPlots` provides an extension of the solver with several plotting functions, e. g., for visualizing MGRIT cycles or the distribution of time points across processes; example plots can be found in Section 4.1.

3.3 | Basic usage of PyMGRIT

In this section, we describe how to solve a simple application problem and we explain the structure of a typical main routine of an application code that uses `PyMGRIT`. For a simple application problem, we choose the simplest case of a scalar ODE. More precisely, our goal is solving Dahlquist's test problem,

$$u' = \lambda u \quad \text{in } (0, 5] \tag{3}$$

with $u(0) = 1$ and $\lambda = -1$. This and other application problems are included in `PyMGRIT`. For an overview of the applications included, see the online documentation. It provides a detailed tutorial and many examples for basic and advanced usage of `PyMGRIT`.
The main routine of an application code that uses `PyMGRIT` to solve Dahlquist's test problem is presented in Listing 2. The program uses a two-level MGRIT algorithm to solve Dahlquist's test problem in the time interval $[0, 5]$ with 101 equidistant time points. The 101 time points are composed of one point for the start time $t = 0$ and 100 other time points. Note that one point for the start time is always included in the time interval in `PyMGRIT`. The structure of a main routine usually consists of three steps. First, the problem is created. Then, a multigrid hierarchy is constructed for this problem. Finally, the problem is solved using the MGRIT algorithm. In our example, for the first step, an instance of `PyMGRIT`'s class `Dahlquist` is created in line 2 that describes the fine problem. The time domain is passed to the problem class by using the parameters `t_start` and `t_stop` for specifying the time interval bounds and the parameter `nt` for the number of time steps. Afterwards, for the second step, a multilevel hierarchy is constructed in line 5, based on the problem instance `dahlquist`. Here, the auxiliary function `simple_setup_problem` is used

```
# Create Dahlquist's test problem with 101 time steps in the interval [0, 5]
dahlquist = Dahlquist(t_start=0, t_stop=5, nt=101)

# Construct a two-level multigrid hierarchy for the test problem using a coarsening factor of 2
dahlquist_multilevel_structure = simple_setup_problem(problem=dahlquist, level=2, coarsening=2)

# Set up the MGRIT solver for the test problem and set the solver tolerance to 1e-10
mgrit = Mgrit(problem=dahlquist_multilevel_structure, tol=1e-10)

# Solve the test problem
info = mgrit.solve()
```

Listing 2: Typical main routine of an application code that uses PyMGRIT.

and a two-level hierarchy with a coarsening factor of two is chosen. This results in a coarse level with 51 points in time. Note that PyMGRIT offers several ways to pass the time domain information to a problem class, as well as for creating a multilevel hierarchy; see the documentation for more details. For the third step, the MGRIT solver for the test problem is set up in line 8 as an instance of PyMGRIT's core class Mgrit using the multilevel object dahlquist_multilevel_structure. Furthermore, the halting tolerance is set to $1e-10$. Finally, the problem is solved by calling the solve routine of the solver mgrit in line 11. The solver returns some statistical information about the run in the dictionary info.

3.4 | Developer view

3.4.1 | Application structure

PyMGRIT is based on four different types of classes:

- Solver: the solver class provides the implementation of the MGRIT algorithm.
- Vector: vector classes contain the solution of a single point in time. Every vector class must inherit from PyMGRIT's core Vector class.
- Application: application classes contain information about the problem we want to solve. Every application class must inherit from PyMGRIT's core Application class.
- GridTransfer: grid transfer classes contain information about the transfer of spatial grids between consecutive MGRIT levels. Every grid transfer class must inherit from PyMGRIT's core GridTransfer class.

The three types of classes Vector, Application and GridTransfer are all based on abstract super classes. These classes independently create some structures that are valid for each type of class and also ensure that all necessary member variables and member functions exist in the respective child classes. A developer who wants to create and solve a problem that is not included in the PyMGRIT package usually only has to specify parts of the four classes. In most cases it is sufficient to write a vector class and an application class. The grid transfer class is primarily needed for the additional feature of spatial coarsening and, if spatial coarsening is not used, it is automatically created by the solver. For the most part, the solver class can be used without modifications, but changes are possible; see the documentation for examples.

3.4.2 | From time-stepping to PyMGRIT

Listing 3 shows a typical time-stepping code for solving problem 2 discretized by backward Euler on a temporal mesh with 101 points. The code consists of three components: first, the initial condition value = 1 is set in line 2. The variable value is further used to store the propagated solution at the current time. The second component consists of the time information in lines 3 and 4. The temporal grid contains $nt = 101$ points in the time interval $[0, 5]$ and is created using the Numpy function linspace. In the last step, we iterate over all points in time and apply the time integration in form of backward Euler. In summary, we have as components a variable that contains the solution at a point in time, time information belonging to the problem, and the time-integration loop.

```
    constant_lambda = -1 # set lambda to -1
    value = 1  # initial solution value
    nt = 101  # number of time points
    t = np.linspace(0, 5, nt)  # time points: nt evenly spaced numbers in interval [0,5]
    # backward Euler time integration
    for i in range(1, nt):
        value = 1 / (1 - (t[i] - t[i - 1]) * constant_lambda) * value
```

Listing 3: Example timestepping code for problem 3, discretized by backward Euler on a temporal mesh with 101 points.

```
class VectorDahlquist(Vector):
    def __init__(self, value):
        super().__init__()
        self.value = value

    def __add__(self, other):
        return VectorDahlquist(self.get_values() + other.get_values())

    def __sub__(self, other):
        return VectorDahlquist(self.get_values() - other.get_values())
```

Listing 4: Vector class for Dahlquist's test equation. Note: The definition of the class is not complete, the member functions `set_values`, `get_values`, `clone`, `clone_zero`, `clone_rand`, `norm`, `pack`, and `unpack` are not shown.

To implement these components in PyMGRIT, we first write a vector class `VectorDahlquist`, which stores the solution at a point in time and inherits from the PyMGRIT core class `Vector`. Therefore, we create a member variable `value`, which contains the solution of a point in time. Furthermore, the following member functions have to be implemented: `set_values`, `get_values`, `clone`, `clone_zero`, `clone_rand`, `__add__`, `__sub__`, `norm`, `pack`, and `unpack`. The function `set_values` receives data values and overwrites the values of the vector data and `get_values` returns the vector data. The function `clone` clones the object, `clone_zero` returns a vector object initialized with zeros, and `clone_rand` similarly returns a vector object initialized with random data. The functions `__add__`, `__sub__`, and `norm` define the addition and subtraction of two vector objects and the norm of a vector object, respectively. Finally, the functions `pack` and `unpack` define the data to be communicated and how data is unpacked after receiving it. The definition of the class `VectorDahlquist` with implementations of the constructor and of the functions `__add__`, `__sub__` is shown in Listing 4. The implementation of the other functions is straightforward and not specified in detail here; please refer to the documentation for more information.
Second, we write an application class `Dahlquist` which contains information about the problem we want to solve. This class contains information about the time grid and the step function and is shown in Listing 5. The time information is automatically provided by the PyMGRIT core class `Application`, from which every PyMGRIT application must inherit from; for more information see the tutorial. The function `step` must be defined and contains the time integration routine, which is the same as in Listing 3 except for names and accesses. To compute the new solution, the function receives as parameters the solution of the previous time point, `u_start`, as well as the start point and the end point of the time integration step, `t_start` and `t_stop`, respectively. Furthermore, two mandatory member variables, `vector_template` and `vector_t_start`, must be created in the application class. The variable `vector_template` stores an instance of the corresponding Vector class, i. e., the `DahlquistVector` class, and `vector_t_start` defines the initial condition using the same class. This is all we need for our test problem. We can now use PyMGRIT to solve our problem as described in Listing 2.

4 | NUMERICAL EXPERIMENTS

In this section, we present several examples for using PyMGRIT to run simulations with MGRIT. Each example has its own focus and highlights different aspects of the framework. The first example gives an overview of runs with different variants of PyMGRIT's MGRIT algorithm and shows some plots generated by PyMGRIT. The goal of this example is to provide a better understanding of the algorithm. The second example shows how PyMGRIT can dramatically reduce the simulation time of an existing realistic application. As an example, a two-dimensional electrical machine is used, where a single time step calls the GetDP[38–40]

```
class Dahlquist(Application):
    def __init__(self, constant_lambda = -1, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.constant_lambda = constant_lambda
        self.vector_template = VectorDahlquist(0) # Set the data structure for any time point
        self.vector_t_start = VectorDahlquist(1) # Set the initial condition

    # Time integration routine
    def step(self, u_start: VectorDahlquist, t_start: float, t_stop: float) ->
    VectorDahlquist:
        return VectorDahlquist(1 / (1 - (t_stop - t_start) * self.constant_lambda ) * u_start
    .get_values())
```

Listing 5: Application class for Dahlquist's test equation.

solver. This example also uses spatial coarsening to further reduce the simulation time. The third example demonstrates the coupling of `PyMGRIT` and `PETSc` and shows space-time parallel results. The tests of all three examples were performed on an Intel Xeon Phi Cluster consisting of four 1.4 GHz Intel Xeon Phi processors.

4.1 | Using PyMGRIT for implementing various MGRIT variants

In the first example, we compare several variants of the MGRIT algorithm for the one-dimensional heat equation,

$$u_t - a u_{xx} = b(x,t) \quad \text{in } [0,1] \times [0,2],$$

with thermal conductivity $a = 1$, right-hand side $b(x,t) = -\sin(\pi x)(\sin(t) - \pi^2 \cos(t))$, homogeneous Dirichlet boundary conditions in space, and subject to the initial condition $u(x,0) = \sin(\pi x)$.
The problem is discretized using second-order central finite differences with 1025 degrees of freedom in space and on an equidistant time grid with 1024 intervals using backward Euler. We choose five-level MGRIT algorithms with coarsening factor $m = 4$ and consider the following variants:

1. V-cycles with FCF-relaxation,
2. V-cycles with $FCFCF$-relaxation,
3. F-cycles with F-relaxation,
4. F-cycles with FCF-relaxation, and
5. F-cycles with $FCFCF$-relaxation.

For all variants, the stopping tolerance is set to 10^{-7} and a random initial guess is used. Note that we are not considering a five-level V-cycle with F-relaxation, since a multilevel setting with only F-relaxation generally does not provide an optimal algorithm[1]. The code for the example can be found in the `examples/toms` directory of `PyMGRIT`.
Figure 9 shows the runtimes and number of iterations of the `PyMGRIT` simulations. The left figure presents the runtime as a function of the number of processes for up to 64 processes and the right figure shows the number of iterations required to reach the halting tolerance for the different MGRIT variants. Comparing only the number of iterations, we see that iterations decrease when stronger (and, thus, more expensive) relaxation schemes are used. Similarly, an F-cycle typically requires fewer iterations than a V-cycle with the same relaxation scheme due to the extra work in each iteration. As an example, the number of iterations for the F-cycle with F-relaxation is about one and a half times the number of iterations for with F-cycle variant with FCF-relaxation. Using the even stronger $FCFCF$-relaxation, the number can be reduced further, but at a smaller scale.
However, the choice of the relaxation scheme and the cycle type affects the cost per iteration. Looking at the runtimes of V-cycles, the runtime of the variant with $FCFCF$-relaxation is always higher than that of the variant with FCF-relaxation, although fewer iterations are required for the stronger relaxation scheme. Looking at F-cycles, we see that runtimes are higher than those of V-cycles for the same relaxation scheme, although iteration counts are smaller. Additionally, F-cycles scale worse than V-cycles.
Figure 10 shows three sample plots that can be automatically generated by the `PyMGRIT` class `MgritWithPlots`. Shown are the plots for the V-cycle variant with FCF-relaxation and using four processes in time. The diagram on the left shows the residual

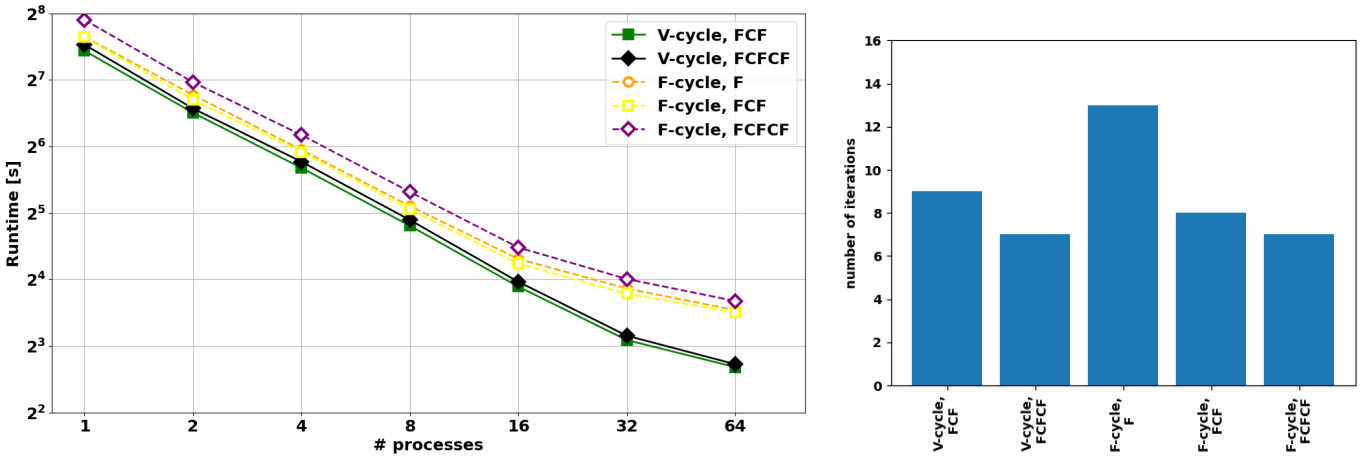

FIGURE 9 Results of five MGRIT variants in terms of runtime (left) and number of iterations (right). The runtime results are shown for using up to 64 processes for parallelizing only in time.

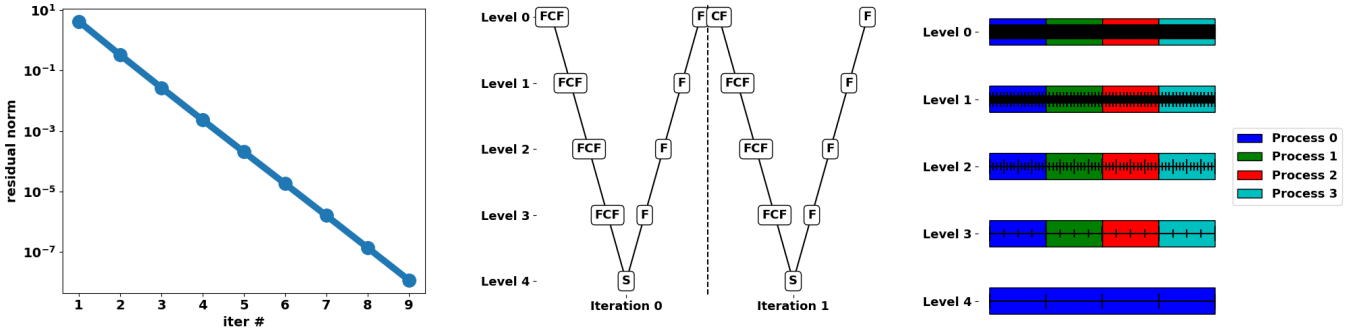

FIGURE 10 Plots provided by `PyMGRIT`'s class `MgritWithPlots` for solving the one-dimensional heat equation using five-level MGRIT V-cycles with FCF-relaxation and using four processes in time. While the convergence plot (left) requires running a full `PyMGRIT` simulation, the plots visualizing the cycling strategy (middle) and the distribution of time points across processes (right) can be created directly after setting up the MGRIT solver.

norm after each MGRIT iteration of a `PyMGRIT` simulation. The other two plots in Figure 10 can be created immediately after initializing the `MgritWithPlots` object. The middle plot visualizes the MGRIT cycle and the right plot presents the distribution of time points across processes for all MGRIT levels.
The figures demonstrate how easy it is to visualize different aspects of `PyMGRIT`'s MGRIT algorithm. More examples of how `PyMGRIT` can be used to create diagrams, e. g., plotting the (space-)time solution of a problem, can be found in the documentation.

4.2 | Time parallelism with PyMGRIT and GetDP

In this example, we apply `PyMGRIT`'s MGRIT algorithm to a simulation of a two-dimensional electrical machine, whereby the solution of the spatial problem at each time step is executed by the external `GetDP` solver. The example demonstrates how an existing time integration routine from a complex simulation code can be integrated into the parallel `PyMGRIT` framework, enabling a dramatic reduction of the simulation time. This section summarizes the results presented in [7]; more details about the simulation can be found in [7] and all files are available in `PyMGRIT`'s folder `examples/induction_machine`.
The standard approach for the simulation of electrical machines is given by the so-called eddy current problem, a simplification of Maxwell's equations in which the displacement current is neglected, and is defined in terms of the unknown magnetic vector

potential $\mathbf{A} : \Omega \times (t_0, t_f] \to \mathbb{R}^3$ as

$$\sigma \partial_t \mathbf{A} + \nabla \times (\nu(\cdot) \nabla \times \mathbf{A}) = \mathbf{J}_s \qquad \text{in } \Omega \times (t_0, t_f], \tag{4a}$$

$$\mathbf{n} \times \mathbf{A} = 0 \qquad \text{on } \partial\Omega, \tag{4b}$$

$$\mathbf{A}\left(\mathbf{x}, t_0\right) = \mathbf{A}_0(\mathbf{x}), \tag{4c}$$

with spatial domain Ω, consisting of rotor, stator, and the air gap in between, time interval $(t_0, t_f]$ and Dirichlet boundary conditions. The geometry is encoded by the scalar electrical conductivity $\sigma(\mathbf{x}) \geq 0$ and the (nonlinear) magnetic reluctivity $\nu(\mathbf{x}, |\nabla \times \mathbf{A}|) > 0$. Three ($n_s = 3$) homogeneously distributed stranded conductors[41] are modeled by the source current density

$$\mathbf{J}_s = \sum_{s=1}^{n_s} \boldsymbol{\chi}_s i_s,$$

with winding functions $\boldsymbol{\chi}_s : \Omega \to \mathbb{R}^3$ and currents $i_s : (t_0, t_f] \to \mathbb{R}^3$. An attached electrical network provides a connection between the voltage $v_s(t)$, $s = 1, 2, 3$, and the so-called flux-linkage.

To consider the rotation of the rotor, the problem is extended by an additional equation of motion,

$$\omega(t) = \frac{d\theta(t)}{dt}, \qquad t \in (t_0, t_f], \tag{5a}$$

$$I \frac{d^2\theta}{dt^2} + C \frac{d\theta}{dt} + \kappa\theta = T_{\text{mag}}(\mathbf{A}) \quad \text{in } (t_0, t_f], \tag{5b}$$

$$\theta\left(t_0\right) = \theta_0, \tag{5c}$$

$$\omega(t_0) = \omega_0, \tag{5d}$$

where the torque T_{mag} defines the mechanical excitation, ω is the angular velocity of the rotor, I denotes the moment of inertia, and C and κ are the fricition and torsion coefficients, respectively. The moving band approach[42] is used to model the movement of the mesh.

We reduce the three-dimensional (3D) domain Ω into a two-dimensional (2D) domain $\Omega_{2D} \subset \mathbb{R}^2$ in the x, y-plane and discretize the problem (4), combined with (5), in space using linear finite elements with n_a degrees of freedom. This yields a system of equations of the form

$$M\mathbf{u}'(t) + K(\mathbf{u}(t))\mathbf{u}(t) = \mathbf{f}(t), \quad t \in (t_0, t_f] \tag{6a}$$

$$\mathbf{u}(t_0) = \mathbf{u}_0, \tag{6b}$$

with unknown $\mathbf{u}^\top = [\mathbf{a}^\top, \mathbf{i}^\top, \theta, \omega] : (t_0, t_f] \to \mathbb{R}^n$. At one point in time $t \in (t_0, t_f]$, the solution $\mathbf{u}(t) \in \mathbb{R}^n$ consists of the magnetic vector potential $\mathbf{a}(t) \in \mathbb{R}^{n_a}$, the currents of the three phases $\mathbf{i}(t) \in \mathbb{R}^3$, the rotor angle $\theta(t) \in \mathbb{R}$, and the angular velocity of the rotor $\omega(t) \in \mathbb{R}$. Note that problem (6), due to the presence of non-conducting materials, consists of differential-algebraic equations (DAEs) of index-1[43, 44].

We use the multi-slice finite element model "im_3_kw"[45] of an induction machine for modeling the semi-discrete problem (6). More precisely, a modified version of the machine[46] supplied by a three-phase pulse width modulated voltage source of 20 kHz is considered. We refer to[7] for more details. Gmsh[47, 48] is used to construct the mesh representation of the model and a hierarchy of two nested meshes is considered to allow for spatial coarsening. The fine mesh Ω_1, consisting of $n_a = 17{,}496$ degrees of freedom, is constructed by refining the coarser mesh Ω_2 with $n_a = 4449$ degrees of freedom. The time step routine calls the `GetDP` solver, which implements the time integration using backward Euler.

We apply five different MGRIT variants to the simulation, choosing Ω_1 as the spatial grid, a time step $\Delta t = 2^{-20}$ and $N_t = 10{,}753$ time steps, resulting in a final time $t_f \approx 0.01$ s. For all MGRIT variants, we choose a convergence criterion based on the relative change of joule losses at C-points of two consecutive iterations. The algorithm stops if the maximum norm of the relative difference of two successive iterations is less than 1%. The following MGRIT variants are chosen: a two-level MGRIT V-cycle with F-relaxation, a five-level MGRIT V-cycle with FCF-relaxation and a five-level MGRIT F-cycle with FCF-relaxation, whereby both five-level variants are applied with and without spatial coarsening. Further, a non-uniform temporal coarsening strategy with coarsening factor 42 on the first level and, for the multilevel variants, a coarsening factor of four on all coarse levels is chosen.

Figure 11 shows the runtimes for the five different MGRIT variants as a function of the number of processes and, for reference purposes, the runtime for sequential time stepping on one processor. Thereby, the dashed line represents the runtime results for sequential time stepping, which is about five days, solid lines represent the MGRIT variants without spatial coarsening and dotted

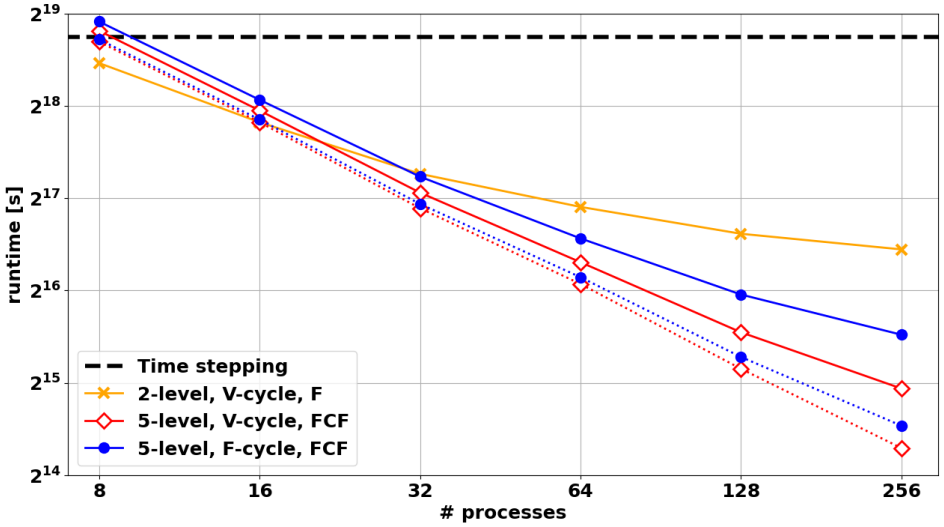

FIGURE 11 Runtimes of five different MGRIT variants and sequential time stepping for the nonlinear electrical machine "im_3_kw". Solid lines represent variants without spatial coarsening and dotted lines with spatial coarsening. For reference purposes, the dashed line shows the runtime for sequential time stepping on one processor.

lines with spatial coarsening. While the multilevel variants require about eight processes to achieve the same runtime results as the sequential method, the two-level variant with eight-way parallelism achieves a runtime of about four days. However, the multilevel variants show better strong scaling, such that the multilevel variants and the two-level variant using 16 processes have approximately the same runtime. Increasing the number of processes to 256, the runtime of the fastest multilevel variant is about 5.5 hours, which corresponds to a speedup of about 22 compared to sequential time stepping.

4.3 | Space-time parallelism with PyMGRIT and PETSc

In the two previous examples, only parallelization in time was considered. In the last example, we show results for space-time parallel runs with `PyMGRIT`. There are several approaches for extending `PyMGRIT` with spatial parallelism, e. g., the use of external libraries that take care of spatial parallelism. Two examples, the coupling of `PyMGRIT` with `Firedrake`[23] and with `PETSc`[49], are available in the online documentation and in the repository. In this example, we consider the coupling of `PyMGRIT` with `PETSc` via the Python package `petsc4py`[34], which allows access to `PETSc` data types, solvers and MPI-based parallization in space without having to use lower-level programming languages like Fortran or C++. Note that `petsc4py` is not automatically installed with the installation of `PyMGRIT`, but can be easily installed using `pip`. The code for the example can be found in the `examples/toms` directory.

To demonstrate space-time parallelization we choose a standard example of parallel-in-time integration methods[17]. More precisely, we choose the the forced 2D heat equation,

$$u_t - \Delta u = b(x, y, t) \quad \text{in } [0, 1] \times [0, 1] \times (t_0, t_f],$$

with initial condition $u(x, y, t_0) = u_0(x, y)$, homogeneous Dirichlet boundary conditions, and forcing term $b(x, y, t)$ such that the exact solution is given by

$$u(x, y, t) = \sin(2\pi x) \sin(2\pi y) \cos(t) \quad \text{in } [0, 1] \times [0, 1] \times [t_0, t_f].$$

The problem is discretized using standard central finite differences in space with $N_x = 129^2$ degrees of freedom. In time, the problem is discretized on a grid with $N_t = 16{,}385$ points using backward Euler.

We apply two five-level MGRIT variants to the problem, a V-cycle with FCF-relaxation and an F-cycle with F-relaxation. Both variants use a coarsening strategy with varying coarsening factors $m_1 = 32$, $m_2 = 16$, $m_3 = 4$, $m_4 = 4$, i. e., coarsening between the finest and the first coarse level is applied using a factor of 32, then a factor of 16 is used, and so on. The stopping criterion for both variants is based on the discrete 2-norm of the space-time residual with a tolerance of 10^{-7}. For both variants, the nested iteration strategy is used to get an improved initial guess.

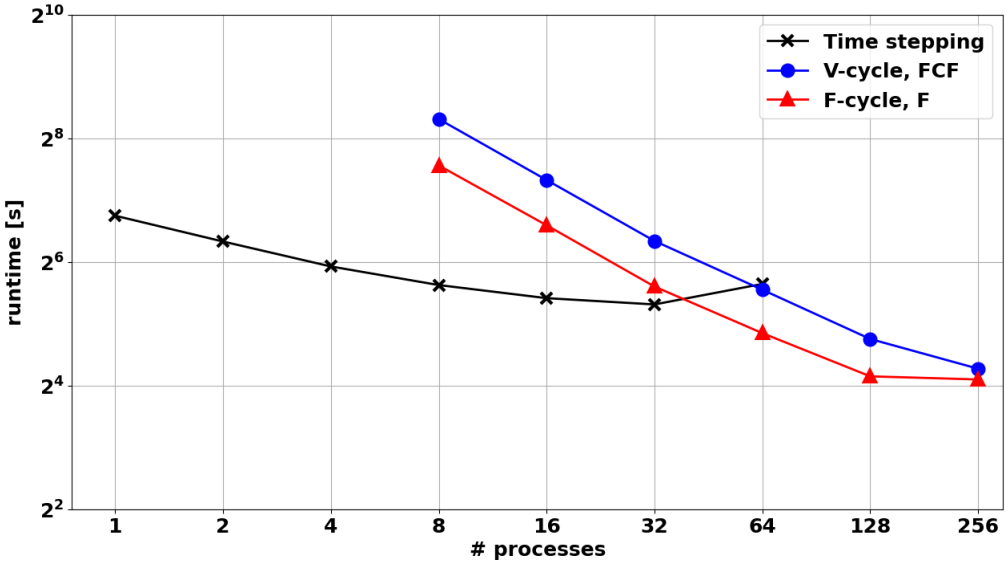

FIGURE 12 Strong scaling results for two five-level MGRIT variants and space-parallel time-stepping. For the multi-level variants, four processes in space are used, i. e., 16 processes correspond to four processes in time and four processes in space.

For the implementation of the `PyMGRIT` application and vector classes for the 2D heat equation, `PETSc` structures can be used thanks to the `PETSc` backend, such that the vector class uses DMDA global vectors as data structure to store the solution at a point in time and the time-stepping method in the application class uses the linear Krylov space (KSP) solver of `PETSc`. Then, space-time parallelism is easily achieved by partitioning the `MPI_COMM_WORLD` communicator into two communicators: one time and one space communicator.

Figure 12 presents strong scaling results for the two MGRIT variants and additionally for a time stepping algorithm, with parallelization only in space. The time-stepping method achieves the fastest runtime for 32 processes in space, so this would be the optimal value to add time parallelism. However, due to the limited number of cores on the cluster, we select only four processes in space for the two MGRIT variants, such that 64 processes in the plot correspond to using four processes for spatial parallelization and 16 processes in time for the MGRIT variants, i. e., a total of $4 * 16 = 64$ processes. Note, that a larger number of processes in space could bring further improvement. With a small number of processes, time stepping is faster than the multi-level variants, but the MGRIT variants show significantly better results with increasing numbers of processes, whereby the crossover point is reached at approximately 64 processes. For 256 processes the speedup of the fastest MGRIT variant is about three compared to space-parallel time-stepping with 32 processes. This could be improved by using even more processes in space and time.

5 | CONCLUSION AND OUTLOOK

This paper introduces the Python framework `PyMGRIT`, which implements the parallel-in-time method multigrid-reduction-in-time (MGRIT). The `PyMGRIT` framework allows the application of the MGRIT algorithm without having to worry about implementation details, parallel communication and so forth, as well as easy prototyping of new variants of the algorithm, and (space-)time parallel simulations using MPI. In addition to the Python code, whose functionality is guaranteed by a continuous integration environment with automated serial and parallel tests, the framework also offers extensive documentation with a quickstart code example, a tutorial, and many examples demonstrating various features of `PyMGRIT`. Together with a simple installation and many pre-implemented ordinary and partial differential equations, the documentation provides an easy start into working with the package. Advanced usage is described in a separate section of the documentation, and more information about `PyMGRIT`'s core classes and functions can be found in the API documentation, which is automatically generated from Python comments. By coupling `PyMGRIT` with other libraries, such as `Firedrake` or `PETSc`, the time parallelism provided by the package can be combined with spatial parallelism for space-time parallel simulations.

The PyMGRIT package already offers many features; however, there are still many possible extensions, open tasks, and new directions. In the following we summarize possible future work. Due to its structure and iterative nature, the MGRIT algorithm allows for a large number of variations and adaptations, e. g., considering different cycle types, applying different relaxation strategies on different MGRIT levels or for different MGRIT iterations, and so forth. While many of these MGRIT variants are already implemented in the PyMGRIT package, there are still some extensions, such as adaptive time stepping, that are not implemented yet. Furthermore, for nonlinear problems, options such as the additional storage of an auxiliary vector[2] that provides an improved starting solution for nonlinear solvers, are attractive extensions.
Another area is the improvement of the required storage space. The MGRIT algorithm, as implemented in PyMGRIT requires more memory than sequential time stepping, because the solution is stored for multiple points in time. There are different strategies to reduce the memory costs, e. g., storing the solution only at C-points, but these are not implemented yet and are topics for future work.
PyMGRIT contains examples of coupling the package with Firedrake, PETSc and GetDP, but of course many other powerful libraries exist. Coupling with more libraries could increase the number of applications for the package and make it easier for users to implement their specific problem. We mention here the libraries FENiCS[50] and PyClaw[51], which both provide a wide range of space-specific algorithms.

ACKNOWLEDGEMENTS

The authors are grateful to Jacob Schroder and Robert Falgout for many helpful conversations and insightful comments regarding the implementation of the MGRIT algorithm, and would like to thank Ryo Yoda for sharing his experiences of using PyMGRIT that led to improvements of the package. The authors also acknowledge the support by the BMBF in the framework of project PASIROM (grant number 05M18PXB).

References

1. Falgout RD, Friedhoff S, Kolev T, MacLachlan S, and Schroder JB. Parallel time integration with multigrid. SIAM Journal on Scientific Computing. 2014;**36**(6):C635–C661.

2. Falgout RD, Manteuffel T, O'Neill B, and Schroder JB. Multigrid reduction in time for nonlinear parabolic problems: a case study. SIAM Journal on Scientific Computing. 2017;**39**(5):S298–S322.

3. Falgout RD, Katz A, Kolev T, Schroder JB, Wissink A, and Yang UM. 2015. *Parallel Time Integration with Multigrid Reduction for a Compressible Fluid Dynamics Application*. . Lawrence Livermore National Laboratory.

4. Lecouvez M, Falgout RD, Woodward CS, and Top P. A parallel multigrid reduction in time method for power systems. In: 2016 IEEE Power and Energy Society General Meeting (PESGM); 2016. p. 1–5.

5. Schroder JB, Falgout RD, Woodward CS, Top P, and Lecouvez M. Parallel-in-Time Solution of Power Systems with Scheduled Events. In: 2018 IEEE Power Energy Society General Meeting (PESGM); 2018. p. 1–5.

6. Friedhoff S, Hahne J, and Schöps S. Multigrid-reduction-in-time for Eddy Current problems. PAMM. 2019;**19**(1):e201900262.

7. Bolten M, Friedhoff S, Hahne J, and Schöps S. Parallel-in-Time Simulation of an Electrical Machine using MGRIT; 2019.

8. Friedhoff S, Hahne J, Kulchytska-Ruchka I, and Schöps S. Exploring Parallel-in-Time Approaches for Eddy Current Problems. In: Faragó I, Izsák F, and Simon PL, editors. Progress in Industrial Mathematics at ECMI 2018. vol. 30 of The European Consortium for Mathematics in Industry. Berlin: Springer; 2020. .

9. Howse A, De Sterck H, Falgout RD, MacLachlan S, and Schroder JB. Parallel-In-Time Multigrid with Adaptive Spatial Coarsening for The Linear Advection and Inviscid Burgers Equations. SIAM Journal on Scientific Computing. 2019;**41**(1):A538–A565.

10. De Sterck H, Falgout RD, Friedhoff S, Krzysik OA, and MacLachlan SP. Optimizing MGRIT and Parareal coarse-grid operators for linear advection; 2019.

11. Günther S, Ruthotto L, Schroder JB, Cyr EC, and Gauger NR. Layer-Parallel Training of Deep Residual Neural Networks. SIAM Journal on Mathematics of Data Science. 2020;**2**(1):1–23.

12. Lions JL, Maday Y, and Turinici G. Résolution d'EDP par un schéma en temps "pararéel". Comptes Rendus de l'Académie des Sciences Série I Mathématique. 2001;**332**(7):661–668.

13. Emmett M, and Minion M. Toward an efficient parallel in time method for partial differential equations. Communications in Applied Mathematics and Computational Science. 2012;**7**(1):105–132.

14. Dutt A, Greengard L, and Rokhlin V. Spectral deferred correction methods for ordinary differential equations. BIT Numerical Mathematics. 2000 Jun;**40**(2):241–266.

15. Christlieb AJ, Macdonald CB, and Ong BW. Parallel High-Order Integrators. SIAM Journal on Scientific Computing. 2010;**32**(2):818–835.

16. Ong BW, Haynes RD, and Ladd K. Algorithm 965: RIDC Methods: A Family of Parallel Time Integrators. ACM Transactions on Mathematical Software. 2016 Aug;**43**(1).

17. Gander MJ. 50 years of Time Parallel Time Integration. In: Multiple Shooting and Time Domain Decomposition. Springer; 2015. p. 69–113.

18. LLNL. Website for XBraid. https://www.llnl.gov/casc/xbraid, Online; accessed June 30, 2020; 2020.

19. LLBL. Website for PFASST codes. https://pfasst.lbl.gov/codes, Online; accessed May 11, 2020; 2020.

20. Martin Schreiber. Website for `SWEET`. https://schreiberx.github.io/sweetsite/, Online; accessed May 11, 2020; 2020.

21. Gander MJ, and Güttel S. PARAEXP: A Parallel Integrator for Linear Initial-Value Problems. SIAM Journal on Scientific Computing. 2013;**35**(2):C123–C142.

22. Hunter JD. Matplotlib: A 2D Graphics Environment. Computing in Science Engineering. 2007;**9**(3):90–95.

23. Rathgeber F, Ham DA, Mitchell L, Lange M, Luporini F, Mcrae ATT, et al. Firedrake: Automating the Finite Element Method by Composing Abstractions. ACM Transactions on Mathematical Software. 2016 Dec;**43**(3).

24. Nielsen AS, Brunner G, and Hesthaven JS. Communication-aware adaptive Parareal with application to a nonlinear hyperbolic system of partial differential equations. Journal of Computational Physics. 2018;**371**:483–505.

25. Falgout RD, Lecouvez M, and Woodward CS. A parallel-in-time algorithm for variable step multistep methods. Journal of Computational Science. 2019;**37**:101029, 12.

26. Brandt A. Multi-level adaptive solutions to boundary-value problems. Math Comp. 1977;**31**(138):333–390.

27. Kronsjö L, and Dahlquist G. On the design of nested iterations for elliptic difference equations. Nordisk Tidskr Informationsbehandling (BIT). 1972;**12**:63–71.

28. Kronsjö L. A note on the "nested iterations" methods. Nordisk Tidskr Informationsbehandling (BIT). 1975;**15**(1):107–110.

29. Hahne J, and Friedhoff S. Github repository for `PyMGRIT`. https://github.com/pymgrit/pymgrit, Online; accessed June 30, 2020; 2020.

30. Hahne J, and Friedhoff S. Documentation for `PyMGRIT`. https://pymgrit.github.io/pymgrit/, Online; accessed June 30, 2020; 2020.

31. Hahne J, and Friedhoff S. PyPI site for `PyMGRIT`. https://pypi.org/project/pymgrit/, Online; accessed June 30, 2020; 2020.

32. van der Walt S, Colbert SC, and Varoquaux G. The NumPy Array: A Structure for Efficient Numerical Computation. Computing in Science Engineering. 2011;**13**(2):22–30.

33. Virtanen P, Gommers R, Oliphant TE, Haberland M, Reddy T, Cournapeau D, et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods. 2020;**17**:261–272.

34. Dalcin LD, Paz RR, Kler PA, and Cosimo A. Parallel distributed computing using Python. Advances in Water Resources. 2011;**34**(9):1124 – 1139. New Computational Methods and Software Tools.

35. Krekel H, Oliveira B, Pfannschmidt R, Bruynooghe F, Laugher B, and Bruhin F. pytest; 2004. Available from: https://github.com/pytest-dev/pytest.

36. Krekel H. Welcome to the tox automation project. https://tox.readthedocs.io/en/latest/, Online; accessed June 30, 2020; 2020.

37. Hahne J, and Friedhoff S. Code coverage for `PyMGRIT`. https://codecov.io/gh/pymgrit/pymgrit, Online; accessed June 30, 2020; 2020.

38. Dular P, Geuzaine C, Henrotte F, and Legros W. A general environment for the treatment of discrete problems and its application to the finite element method. IEEE Transactions on Magnetics. 1998 Sep;**34**(5):3395–3398.

39. Geuzaine C. GetDP: a general finite-element solver for the de Rham complex. PAMM. 2007;**7**(1):1010603–1010604.

40. Dular P, and Geuzaine C. GetDP: A General Environment for the Treatment of Discrete Problems. http://www.getdp.info, Online; accessed June 30, 2020; 2020.

41. Schöps S, Niyonzima I, and Clemens M. Parallel-in-time Simulation of Eddy Current Problems using Parareal. IEEE Transactions on Magnetics. 2018 März;**54**(3).

42. Ferreira da Luz MV, Dular P, Sadowski N, Geuzaine C, and Bastos JPA. Analysis of a permanent magnet generator with dual formulations using periodicity conditions and moving band. IEEE Transactions on Magnetics. 2002 March;**38**(2):961–964.

43. Bartel A, Baumanns S, and Schöps S. Structural Analysis of Electrical Circuits Including Magnetoquasistatic Devices. Applied Numerical Mathematics. 2011 Sep;**61**:1257–1270.

44. Cortes Garcia I, De Gersem H, and Schöps S. A Structural Analysis of Field/Circuit Coupled Problems Based on a Generalised Circuit Element. Numerical Algorithms. 2019 Mar;p. 1–22.

45. Gyselinck J, Vandevelde L, and Melkebeek J. Multi-slice FE modeling of electrical machines with skewed slots-the skew discretization error. Magnetics, IEEE Transactions on. 2001 10;**37**:3233 – 3237.

46. Gander MJ, Kulchytska-Ruchka I, Niyonzima I, and Schöps S. A New Parareal Algorithm for Problems with Discontinuous Sources. SIAM Journal on Scientific Computing. 2019;**41**(2):B375–B395.

47. Geuzaine C, and Remacle JF. Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. International Journal for Numerical Methods in Engineering. 2009;**79**(11):1309–1331.

48. Geuzaine C, and Remacle JF. Gmsh: A three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. http://www.gmsh.info, Online; accessed May 11, 2020; 2020.

49. Balay S, Abhyankar S, Adams MF, Brown J, Brune P, Buschelman K, et al.. 2020. *PETSc Users Manual*. ANL-95/11 - Revision 3.13. Argonne National Laboratory. Available from: https://www.mcs.anl.gov/petsc.

50. Logg A, Mardal KA, Wells GN, et al. Automated Solution of Differential Equations by the Finite Element Method. Springer; 2012.

51. Ketcheson DI, Mandli KT, Ahmadia AJ, Alghamdi A, Quezada de Luna M, Parsani M, et al. PyClaw: Accessible, Extensible, Scalable Tools for Wave Propagation Problems. SIAM Journal on Scientific Computing. 2012 Nov;**34**(4):C210–C231.