Parallel-in-Time Integration with preCICE

Tobias Eppacher

Chair for Scientific Computing
School of Computation, Information and Technology
Technical University of Munich

Parallel-in-Time Integration with preCICE

Tobias Eppacher

Thesis for the attainment of the academic degree

Bachelor of Science (B.Sc.)

at the School of Computation, Information and Technology of the Technical University of Munich.

Examiner:
Univ.-Prof. Dr. Hans-Joachim Bungartz

Supervisor:
Keerthi Gaddameedi, M.Sc. & Benjamin Rodenberg, M.Sc. (hons)

Submitted:
Munich, 31.08.2024

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated.
I have only used the resources given in the list of references.

Munich, 31.08.2024 Tobias Eppacher

Acknowledgements

I would like to express my appreciation to the Chair for Scientific Computing for providing me with the opportunity to work on this thesis.

I also want to thank my advisors, Keerthi and Benjamin, for their guidance and support throughout the thesis. Every meeting with them provided valuable insights and helped me stay on track.

To my family and friends, I can only express my gratitude for their continuous support and understanding. They were always there for me when I needed them.

Abstract

Modern physics simulations are often complex and computationally expensive. A common approach to tackle this problem involves the division of the simulation domain into smaller subdomains, which are solved separately. This domain partitioning also involves a process to recombine the partial solutions to a global one, named *coupling*. Different numerical methods are then available to solve the subproblems. Promising methods for this are parallel-in-time (PinT) methods, which aim to parallelize the time stepping over the temporal domain, with a promising candidate being the spectral deferred correction (SDC)-based PFASST algorithm. A combination of domain partitioning and PinT methods seems to be a promising candidate for efficient simulation parallelization. However, for the successful combination of these methods, it is desired that the interaction of both methods does not alter their error behavior significantly.

This thesis focuses on the analysis of error and convergence of an SDC-based time integrator in the context of domain partitioning. In this context, we implement a solver using SDC and carry out simulations to investigate its error behavior with different settings. For the implementation of the solver, we use the open-source libraries FEniCS and pySDC, of which FEniCS provides spatial discretization using the finite element method (FEM), and pySDC serves as a framework for SDC. For coupled simulations, we use the open-source library preCICE. As scenarios for the simulations, we use multiple adapted versions of the forced heat equation in a two-dimensional domain. We apply the method of manufactured solutions (MMS) to create analytical reference solutions we then use to verify the implementation and measure the conducted simulations' errors. We analyze the error behavior in the generated data and suggest further research.

Contents

1 Introduction

Many problems in Science and Engineering require the solution of initial or boundary value problems to simulate the evolution of a system through time.[5, 13] Such problems can generally be described by ordinary differential equations (ODEs) or partial differential equations (PDEs), where an analytical solution often only exists for specialized scenarios. An example of this would be the famous three-body problem [25]. This leads to the necessity of simulations using numerical methods for generalized scenarios.

When simulations consider multiple physical phenomena, one can speak of the conduction of a *multi-physics simulation*. Depending on the problem, e.g. the domain size and the simulated time interval, these can become very complex and computationally expensive. Additionally, there often are separate software tools available for the simulation of each specific physical phenomenon, but not necessarily for multiple of them at once.

The method of choice is then often a divide-and-conquer approach, where the calculations are separated by physical phenomena, the simulation domain is partitioned into smaller subdomains or even both. In this thesis, we will focus on the latter, *domain partitioning*, which is a common approach to parallelize simulations by solving the subproblems on different processors or even different machines. As simple as the partitioning may seem, the dependence between the subproblems requires a method to recombine their solutions. This process is called *coupling*, and transfers necessary information from one solver to another. From here on, we will refer to a solver who participates in a coupled simulation as a *participant*. In the case of domain partitioning, the coupling can exchange values at the surface between subdomains (surface coupling). The coupling process generally introduces new possibilities for errors, depending on the data exchange method, different time step sizes or spatial resolutions of the participants and the coupling frequency.

Regarding the solving of the subproblems themselves, there are different numerical methods available. Probably the most simple of those methods is classical time-stepping with the explicit Euler method, which is easy to implement and understand, but has severe limitations in terms of stability and accuracy. Both can be improved by using higher-order and implicit methods respectively. The major drawback of implicit schemes of higher order is, that the time steps become computationally expensive. This increases simulation times for high temporal resolution dramatically, as conventional time stepping is inherently serial. PinT methods try to counteract this, as they aim to parallelize the time stepping over the temporal domain. Some prominent examples of PinT algorithms are Parareal [19], Parallel Full Approximation Scheme in Space and Time (PFASST) [10], and Multigrid Reduction in Time (MGRIT) [11]. PFASST is interesting for its use of SDC to create a higher-order method by using low-order time integrators.

Both domain partitioning and PinT methods provide a way to speed up simulations by parallelizing calculations. Therefore the combination of both seems to be a promising candidate for research. In this thesis, we conduct simulations of several scenarios of the forced heat equation in a two-dimensional domain using the SDC method, the basis of the PinT algorithm PFASST. For the implementation of the SDC method, we use the open-source library *pySDC*. To solve a PDE, also spatial discretization is required, which is done using the FEM with the library *FEniCS*. The conducted simulations include monolithic and partitioned cases, to search for potential defects or a lowered convergence rate introduced by the coupling process. The coupling process is implemented using the open-source library preCICE. For the verification of the implementation and to create an error measure, we create predetermined analytical solutions to the heat equation using the method of manufactured solutions (MMS). We then analyze the conducted simulations regarding error and convergence and suggest further research possibilities.

We first provide a summarized introduction to the used algorithms and methods, including SDC, FEM and domain partitioning in Chapter 2. In Chapter 3, the software tools used in this thesis are described. We proceed with the details of the implementation in Chapter 4. The conducted simulations and their results are presented in Chapter 5, with the conclusion and the suggestion of further research possibilities in Chapter 6.

2 Foundations

The major schemes and methods used in this thesis are SDC for time integration and the FEM to solve the spatial part of the problem. To create a coupled simulation, we use domain partitioning and Dirichlet-Neumann coupling as coupling scheme. This chapter aims to give an overview of those methods and an introduction to their mathematical backgrounds.

2.1 Spectral Deferred Correction

The SDC methods were introduced in 2000 by Dutt et al. [9]. The basic idea behind SDC is to construct methods equivalent to high-order implicit schemes by the use of cheaper low-order time integration schemes. This section is aimed to summarize the basic concepts of SDC.

2.1.1 The Picard Integral Equation

Assuming we want to solve an initial value problem of the form

$$\frac{\partial u}{\partial t} = F(u(t), t)$$

The original problem is replaced with the corresponding Picard integral equation, by integration w.r.t. the time t:

$$u(t) = u(t_0) + \int_{t_0}^{t} F(s, u(s))ds \tag{2.1}$$

Given some approximation $u_{approx}(t)$ of the solution, the residual and error functions can be formulated as

$$\epsilon(t) = u(t_0) + \int_{t_0}^{t} F(s, u_{approx}(s))ds - u_{approx}(t) \tag{2.2}$$

$$\delta(t) = u(t) - u_{approx}(t) \tag{2.3}$$

By combining the Functions 2.1, 2.2 and 2.3 we can reformulate the equations to form

$$\delta(t) = \int_{t_0}^{t} [F(s, u_{approx}(s) + \delta(s)) - F(s, u_{approx}(s))]ds + \epsilon(t)$$

Summarizing the integrand in the above equation to

$$G(t, \delta(t)) = F(s, u_{approx}(s) + \delta(s)) - F(s, u_{approx}(s))$$

all these reformulations yield another Picard-like integral equation

$$\delta(t) - \int_{t_0}^{t} G(s, \delta(s))ds = \epsilon(t) \tag{2.4}$$

2.1.2 Picard Iteration

The method with which integral equations, such as 2.1 are solved is called Picard-iteration. It is a fixed-point iteration method, where the constructed series of functions converges to the solution of the integral equation for sufficiently small values of h.

$$u^{[0]}(t) = u_0 \qquad \text{where } u_0 = u(t_0)$$
$$u^{[i+1]}(t) = u_0 + \int_{t_0}^{t} F(s, u^{[i]}(s))ds, \qquad t \in [t_0, t_0 + h]$$

In our case, we never know the function over the full time interval, but at discrete points in time. Those are then used, to replace the exact computation of the integral by a low-order numerical time integration method. For the following example we use the implicit or explicit Euler method. Assume we are given some set of points in time $t_0 < t_1 < \cdots < t_N$ on a time interval $[a, b]$ and an initial value $u_0 = u(t_0)$, for which we want to compute the solution for the integral equation 2.1. Starting from t_0 we iteratively apply explicit or implicit Euler steps given by the following formulas. Note that the use of implicit Euler steps also requires an initial value for the solution at time t_1.

$$u_{n+1} = u_n + h_i F(u_n, t_n), \quad h_i = t_{i+1} - t_i \quad \text{(Explicit Euler Step)}$$

$$u_{n+1} = u_n + h_i F(u_{n+1}, t_{n+1}), \quad h_i = t_{i+1} - t_i \quad \text{(Implicit Euler Step)}$$

Similarly, the Euler steps can be defined for the Picard-like equation 2.4.

$$\delta_{i+1} = \delta_i + h_i G(t_i, \delta(t_i)) + (\epsilon(t_{i+1}) - \epsilon(t_i)) \quad \text{(Explicit Euler Step)}$$

$$\delta_{i+1} = \delta_i + h_i G(t_{i+1}, \delta(t_{i+1})) + (\epsilon(t_{i+1}) - \epsilon(t_i)) \quad \text{(Implicit Euler Step)}$$

The application of such time-stepping yields approximated values for the solution and the error function at all used discrete points in time. These are the discretized solution and error function in the first iteration of the Picard iteration. The error function values can then be used to correct the solution approximation, to produce the next, better approximation. This process is repeated iteratively such that the approximations converge towards the true solution of the integral equation.

2.1.3 Spectral Integration

In the previous subsection, we have seen that Euler steps can be used to approximate the integrals in the Picard iteration. However, the computation of the residual function $\epsilon(t)$, which is also required for the mentioned Euler steps, requires integrals of the right-hand side function $F(u, t)$. Since one of those is required for every discrete time point we will now introduce the concept of spectral integration as an efficient way to calculate those integrals.

Assume we have a function $\phi(t)$ for which we want to calculate the integral over some interval $[a, b]$.

$$\int_a^b \phi(t)dt$$

For this purpose we assume to be given function values $\phi = \{\phi_1, \phi_2, \ldots, \phi_N\}$ on a set of points in time $t_1 < \cdots < t_N$ within the interval $[a, b]$.

We then construct the Lagrange interpolation polynomial $L^N(\phi, t)$ for ϕ using the given function values and support points.

$$L^N(\phi, t) = \sum_{i=1}^{m} \phi(t_i) l_i(t)$$

$$l_i(t) = \prod_{j \neq i} \frac{t - t_j}{t_i - t_j}$$

The original integral can then be approximated by integration of the Lagrange interpolant of ϕ.

$$\int_a^b \phi(t)dt \approx \int_a^b L^N(\phi, t)dt \quad = \int_a^b \sum_{i=0}^{m} \phi(t_i) l_i(t)dt = \sum_{i=1}^{m} \phi(t_i) \int_a^b l_i(t)dt$$

Thus we can represent the original integral as a weighted sum of the integrated basis functions $l_i(t)$, where the weights are the function values $\phi(t_i)$.

Since the computations of the residual at our discrete time points require one integral each, from the same starting time a to different end times $t_1, \ldots, t_N$, we perform those calculations as a single matrix-vector product.

$$S^N \phi = \psi$$

$$\phi = \{\phi(t_1), \ldots, \phi(t_N)\} \in \mathbb{R}^N$$

$$(S^N)_{j,i} = \int_a^{t_j} l_i(t)dt \qquad \text{for } i, j = 1, \ldots, N$$

This multiplication of ϕ with the so-called spectral integration matrix S^N yields a vector ψ. This vector contains the approximated integrals of the function $\phi(t)$ from a to $t_1, \ldots, t_N$. respectively.

By using constant support points, we can use the fact that the Lagrange basis functions only depend on those points, which allows the precomputation and storage of those integrals. Additionally, any definite integral over some arbitrary interval can be written as a 'scaled' version over the interval $[-1, 1]$, by simple variable transformation. This further simplifies the process, such that for fixed distributions of support points within an interval, the corresponding integrals of the basis functions $l_i(t)$ on the interval $[-1, 1]$ can be stored in a lookup table and scaled on demand for our calculations.

It remains the question of what distribution of support points to choose, as it can be expected, that they have a significant influence on the accuracy of the interpolation and integral approximation. The use of equidistant points for example, can lead to numerical instabilities like the Runge phenomenon. To prevent this, Gauss-Legendre or Gauss-Lobatto quadrature nodes are usually a good choice for the support points. For this reason, we will call the support points *nodes* in the following chapters.

2.1.4 The Algorithm

After the introduction of the main components, we can now outline the SDC algorithm using the following definitions.

$$\frac{\partial u}{\partial t} = F(u, t) \quad \text{Problem definition}$$

$s_1, \ldots, s_N$	Gauss-Legendre quadrature nodes on the interval $[a, b]$
$\overline{u_a} = (u_a, u_a, \ldots, u_a)$	N element vector of the initial value of the solution
$u^j = (u^j(s_1), \ldots, u^j(s_N))$	Vector of j-th approximation solution values
$\overline{F}(u^j) = (F(s_1, u^j(s_1)), \ldots, F(s_N, u^j(s_N)))$	Vector of j-th approximation right hand side evaluations

Recall the residual function 2.2. We now replace the exact integral with spectral integration as described in the previous subsection to obtain

$$\sigma(u^j) = S^N \overline{F}(u^j) - u^j + \bar{u_a}$$

where $\sigma(u^j)$ is the vector containing the residual function values for all nodes with the j-th approximation of the solution. The full procedure can be summarized by the following outline:

Algorithm 1 Spectral Deferred Correction

▷ *Initialization* ◁

Compute initial approximation u^0 for all nodes $s_1, \ldots, s_N$ on the interval $[a, b]$ using a low-order time integration method (in our case explicit/implicit Euler, depending on the stiffness of the problem)

▷ *Iterative Corrections* ◁

for $j = 0, \ldots, J$ **do**

(1) Compute the current approximate residual function $\sigma(u^j)$

(2) Compute the error function δ^j by solving the Picard-like equation 2.4 with a low-order method (explicit/implicit Euler for us). The initial value for the error function $\delta^j(a)$ is set to zero.

(3) Update the approximation $u^{j+1} = u^j + \delta^j$

Some remarks on the algorithm:

- One full iteration of the algorithm, where the approximation is updated at all nodes is called a *sweep*. The number of sweeps, J, can be chosen arbitrarily large, where more sweeps produce a more accurate approximation of the true solution. If the true solution is of a temporal degree smaller or equal than the number of nodes N, the interpolation polynomial and the spectral integration is exact, yielding the best possible approximation after a single iteration. Even if this is not the case, it is generally recommended to set an upper limit for the number of iterations, and potentially stop early if the residual function is small enough.
- The choice between explicit or implicit time stepping methods depends on the stiffness of the problem. For non-stiff problems, the explicit Euler method is used, while for stiff problems, the implicit Euler method is used. Note that for implicit Euler steps, the initial value for the solution and the error function at time t_1 is required. Since the error is estimated to be small, 0 can generally be used as an initial value. The initial value for the solution at time t_1 could be estimated using a single explicit Euler step.
- Throughout the algorithm, the initial condition at time a for a time interval is known and the solution value for the end time b is required. Since Gauss-Legendre nodes do not cover the boundaries of the interval, no result for the solution at time b is available. This is no problem, as the interpolating Lagrange polynomial can be evaluated at that point to generate a value. Another solution would be the use of Gauss-Lobatto nodes, which ensure support on the interval boundaries, but provide accurate results up to a lower degree than Gauss-Legendre nodes. This is also the solution our implementation uses.

One of the biggest advantages of this algorithm is its black-box nature. Once implemented, it can be applied to any problem of the form $\frac{\partial u}{\partial t} = F(u, t)$ without further modifications. The only thing that needs to be supplied is the evaluation of the right-hand side function $F(u, t)$ for some given solution approximation at arbitrary points in time. Of course, boundary conditions and initial conditions also have to be provided to ensure a unique solution to the problem.

2.1.5 Multi-Level SDC

Several variants and improvements of this original SDC method have been developed over the years. One of the more significant ones is the extension to Multi-Level Spectral Deferred Correction (MLSDC) [31]. The idea behind this variant is to perform the SDC sweeps on multiple levels of spatial discretizations.

Those levels are then coupled using a FAS (Full Approximation Scheme) correction term, which enhances the accuracy of the coarser levels. This way, some sweeps on the finer levels can be replaced by cheaper sweeps on the coarser levels (due to the coarser spatial grid), which can lead to a significant speedup of the overall algorithm. This approach is very similar to the V-cycles used in multigrid methods.

2.1.6 PFASST

PFASST (Parallel Full Approximation Scheme in Space and Time) [10] is another variant of the SDC method, which aims to parallelize the SDC algorithm in the time dimension. The approach is very similar to the MLSDC method, in the sense that it uses coarsened temporal grids (on the coarser of which spatial coarsening can be performed as well). The main difference is that the PFASST algorithm is designed to run successive corrections on separate time intervals in parallel. Therefore it is also denoted as a parallel-in-time method.

2.2 Finite Element Method

In comparison to ordinary differential equations, partial differential equations require not only temporal but also spatial discretization. This fact has to be acknowledged in our method to calculate the temporal derivative for SDC. There exist several methods for spatial discretization, like Finite Differences, Finite Volumes, and Finite Elements. Especially for problems in physics and engineering, the FEM is a widely used method with applications including heat transfer [1], fluid dynamics [34], structural engineering [27], and many more.

The FEM is based on the idea of discretizing the domain of the PDE into a finite amount of smaller, simpler subdomains, called elements, the namesake of the method. Depending on the dimensions of the problem, these elements can take many different forms, be it lines, triangles, quadrilaterals, tetrahedra and more[1]. The elements themselves are described by a set of nodes, which are used as support points for the solution within one element. In one dimension, these could be the endpoints of a line segment. Depending on the number of nodes, the interpolation can be linear, quadratic, or even higher order. The resulting systems of equations for the solution in the different elements are then combined to form a global system, which can be solved for the global solution of the problem.

2.2.1 The Forced Heat Equation

The results of this thesis are based on tests around the forced heat equation, a PDE that describes the heat distribution in a domain over time, with an additional forcing term f that characterizes its behavior. Therefore this chapter will take said equation as an example to explain the applied concepts.

$$\dot{u} - \nabla^2 u = f \qquad \text{(Forced Heat Equation)}$$
$$\nabla^2 u = \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} + \cdots + \frac{\partial^2 u}{\partial x_n^2} \qquad \text{(Laplacian of } u(\vec{x}, t)\text{)}$$

In this equation $u(\vec{x}, t)$ is the unknown temperature distribution function in the domain, $\dot{u}$ is the derivative w.r.t time t of that function, and f is the forcing term. Additionally, boundary conditions on the domain boundary $\partial\Omega$ are required. For simplicity, we will for now assume homogeneous Dirichlet boundary conditions (BCs), i.e. $u = 0$ on the complete boundary $\partial\Omega$.

[1] A comprehensive list of commonly used elements can be found at [4].

2.2.2 Variational Formulation

After the definition of the PDE and the boundary conditions, the first step is the creation of the variational formulation of the problem. This formulation is derived by multiplying the PDE with a test function and integrating it over the spatial domain. Such a test function is an element of some function space V taylored to the chosen mesh of elements. In a one-dimensional case, an example for this could be a space of hat-functions over the domain.

If we apply this to the forced heat equation, we get the following variational formulation.

$$\int_\Omega \dot{u}v\,dx - \int_\Omega (\nabla^2 u)v\,dx = \int_\Omega fv\,dx \qquad \forall v \in V$$

By applying integration by parts we have

$$\int_\Omega \dot{u}v\,dx + \int_\Omega \nabla u \nabla v\,dx - \int_{\partial\Omega} \frac{\partial u}{\partial n} v\,ds = \int_\Omega fv\,dx \qquad \forall v \in V \tag{2.5}$$

where n is the outwards normal vector on the domain boundary $\partial\Omega$. In this variational formulation, it is necessary for the test function v to be zero on the parts of the boundary where the solution is known.[17]

Using our assumptions about the boundary, i.e. $u = 0$ on $\partial\Omega$, and the fact that v is zero on all parts of the boundary with known values, we can simplify the equation to

$$\int_\Omega \dot{u}v\,dx + \int_\Omega \nabla u \nabla v\,dx = \int_\Omega fv\,dx \qquad \forall v \in V$$

2.2.3 Spatial Discretization

We discretize this variational formulation by finding a function u_h in some trial function space W that satisfies the equation.

$$\int_\Omega \dot{u_h}v\,dx + \int_\Omega \nabla u_h \nabla v\,dx = \int_\Omega fv\,dx \qquad \forall v \in V$$

Generally, the trial function space W and the test function space V can be different, but for our purposes we set $W = V$. Since each function in V can be described as a linear combination of its basis functions, it suffices to show the variational equation holds for those basis functions of V. We denote them as $\phi_i, i = 0, \ldots, n$.

$$\int_\Omega \dot{u_h}\phi_i\,dx + \int_\Omega \nabla u_h \nabla \phi_i\,dx = \int_\Omega f\phi_i\,dx \qquad i = 0, \ldots, n$$

The solution approximation u_h is also an element of V and can therefore as well be described as a linear combination of the basis functions $\phi_j(x)$ and the corresponding time-dependent coefficients $\xi_j(t)$ with $j = 0, \ldots, n$.

$$u_h(x,t) = \sum_{j=0}^{n} \xi_j(t)\phi_j(x)$$

Inserting this ansatz into the variational formulation, we get

$$\begin{aligned}\int_\Omega f\phi_i\,dx = &\sum_{j=0}^{n} \dot{\xi}_j(t) \int_\Omega \phi_j\phi_i\,dx \\ &+ \sum_{j=0}^{n} \xi_j(t) \int_\Omega \nabla\phi_j \nabla\phi_i\,dx \qquad i = 0, \ldots, n\end{aligned}$$

We then use the following notation

$$M_{ij} = \int_\Omega \phi_j \phi_i \, dx$$
$$K_{ij} = \int_\Omega \nabla\phi_j \nabla\phi_i \, dx$$
$$b_i(t) = \int_\Omega f(t)\phi_i \, dx$$

to rewrite the equation as matrix formulation

$$M\dot{\xi}(t) + K\xi(t) = b(t) \tag{2.6}$$

where M is called the mass matrix, K the stiffness matrix.

Normally, the FEM would now use time discretization to create a system of equations that makes use of known coefficients $\xi(t_i)$ to solve for the coefficients (i.e. the solution) at time t_{i+1}. However, in the context of this thesis, we are interested in calculating the temporal derivative of the coefficients, $\dot{\xi}$, out of the known coefficients ξ at time t. By simple reformulation of Equation 2.6 we get a rather short and elegant equation to do so.

$$\dot{\xi}(t) = M^{-1}(b(t) - K\xi(t))$$

Keep in mind, that both, the mass matrix M and the stiffness matrix K are time-independent and can be precomputed, once the domain mesh and used trial and test function spaces are determined.

2.2.4 Dirichlet and Neumann Boundaries

In the previous sections, we assumed homogeneous Dirichlet BCs, i.e. $u = 0$ on the complete boundary $\partial\Omega$. However, in many cases, this assumption is not valid and we need to consider non-zero Dirichlet or Neumann BCs.

For non-zero Dirichlet BCs, we can use the same approach as for homogeneous Dirichlet BCs, by simply letting the test function v be zero on the parts of the boundary where the solution is known.

Different from Dirichlet BCs, Neumann BCs do not specify a known value for the solution, but rather a known rate of change of the solution in the outwards normal direction on the boundary (also called *flux*). To incorporate this into our method, we cannot use the same approach as for Dirichlet BCs but rather have to return to our variational formulation.

Recall Equation 2.5

$$\int_\Omega \dot{u}v \, dx + \int_\Omega \nabla u \nabla v \, dx - \int_{\partial\Omega} \frac{\partial u}{\partial n} v \, ds = \int_\Omega fv \, dx \qquad \forall v \in V$$

This time we assume a Neumann boundary condition $\frac{\partial u}{\partial n} = g$ on the boundary $\partial\Omega$, which can be inserted into the equation.

$$\int_\Omega \dot{u}v \, dx + \int_\Omega \nabla u \nabla v \, dx = \int_\Omega fv \, dx + \int_{\partial\Omega} gv \, ds \qquad \forall v \in V$$

Following the same steps as before, we discretize this equation and get a matrix formulation including Neumann BCs. Note that only the right-hand side vector $b(t)$ is affected.

$$M_{ij} = \int_\Omega \phi_j \phi_i \, dx$$
$$K_{ij} = \int_\Omega \nabla\phi_j \nabla\phi_i \, dx$$
$$b_i(t) = \int_\Omega f(t)\phi_i \, dx + \int_{\partial\Omega} g(t)\phi_i \, ds$$
$$M\dot{\xi}(t) + K\xi(t) = b(t)$$

2.3 Domain Partitioning

The first step when applying domain partitioning to divide a PDE, is to define the subdomains.

In the further context of this thesis, we will use $\Omega = [0, 2] \times [0, 1]$ as the full simulation domain for the forced heat equation. Additionally, we assume Dirichlet BCs on the boundary $\partial\Omega$.

We will divide this domain into two subdomains, $\Omega_{\mathcal{D}}$ and $\Omega_{\mathcal{N}}$ along the line $x = 1$, yielding the left and right half of Ω respectively, as seen in Figure 2.1.

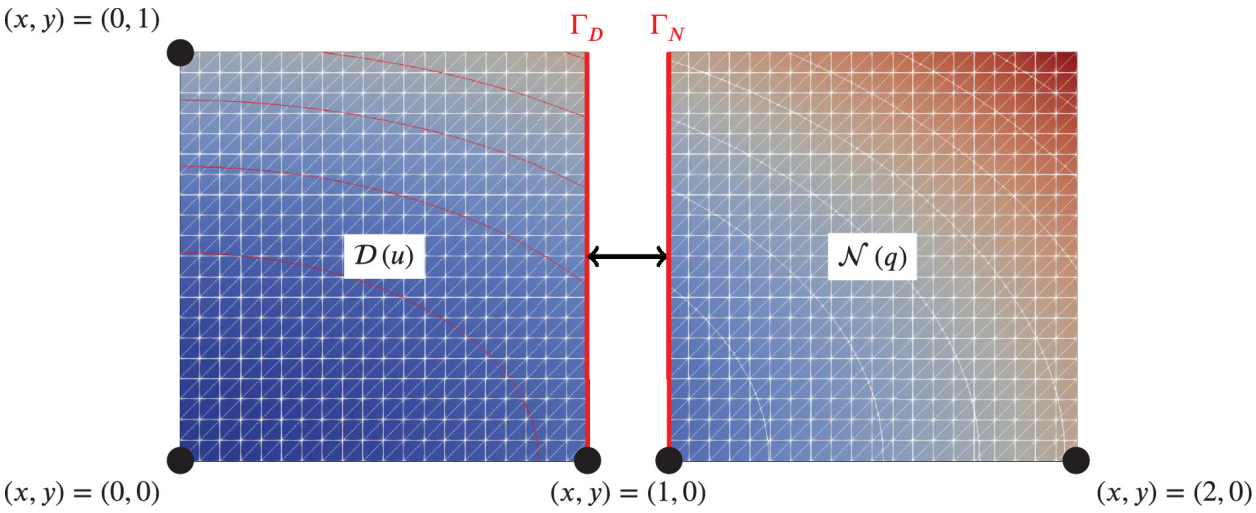

Figure 2.1 The domain Ω partitioned into the subdomains $\Omega_{\mathcal{D}}$ and $\Omega_{\mathcal{N}}$ along the line $x = 1$. The dividing boundary is labeled $\Gamma_{\mathcal{D}}$ and $\Gamma_{\mathcal{N}}$ respectively. This figure was taken from [29].

The surface given by the splitting line segment is part of the boundary for both subdomains and therefore affects and is affected by the computations on both subdomains. For this reason, this surface is called *coupling boundary* and named $\Gamma_{\mathcal{D}}$ and $\Gamma_{\mathcal{N}}$ depending on the current subdomain. The data exchange at this boundary is then part of the coupling process.

2.3.1 Coupling

Since coupling involves the transfer of necessary data between two participants, there exist multiple coupling schemes that can be used, which can be classified in several regards. One can speak of *uni-directional* or *bi-directional* (depending on the direction of the data exchange), *explicit* or *implicit* (single or multiple executions of participants per time step), *parallel* or *serial* (regarding executions of participants) and *surface* or *volume* coupling (whether the exchanged data stems from the surface or a part volume of a domain).[7] All of these classes provide a plethora of possible schemes. The one best suited to a problem has to be determined depending on the definition of the subdomains and the imposed boundary conditions at the coupling boundaries. In our case, we split the full domain into non-overlapping subdomains, which makes surface coupling the most natural choice. We also impose Dirichlet boundary conditions on $\Gamma_{\mathcal{D}}$ and Neumann boundary conditions on $\Gamma_{\mathcal{N}}$. These choices lead to the application of Dirichlet-Neumann coupling.

2.3.2 Dirichlet-Neumann Coupling

The Dirichlet-Neumann coupling scheme is an iterative coupling scheme generally outlined by Algorithm 2.

By this definition, Dirichlet-Neumann coupling can be classified as *bi-directional* and *implicit* as data is exchanged in both directions between the Dirichlet and the Neumann participant and both have to be executed possibly multiple times until the tolerance ϵ is reached. It is also inherently serial, as the

Algorithm 2 Dirichlet-Neumann Coupling

1: Set initial Dirichlet boundary condition on $\Gamma_{\mathcal{D}}$ as some f_0
2: Set $k = 0$
3: **repeat**
4: Compute solution $u_{\mathcal{D}}^{[k]}$ on $\Omega_{\mathcal{D}}$ using f_k as Dirichlet boundary condition
5: Calculate the flux $g_k = \frac{\partial u_{\mathcal{D}}^{[k]}}{\partial n}$ on $\Gamma_{\mathcal{D}}$ (n is the outwards normal vector on $\Gamma_{\mathcal{D}}$)
6: Compute solution $u_{\mathcal{N}}^{[k]}$ on $\Omega_{\mathcal{N}}$ using g_k as Neumann boundary condition
7: Calculate new values for f_{k+1}. In this step, relaxation factors can be used to accelerate the convergence of the scheme. E.g. [24] uses $f_{k+1} = \theta_k u_{\mathcal{N}|\Gamma_{\mathcal{N}}}^{[k]} + (1 - \theta_k) f_k$
8: Set $k = k + 1$
9: **until** $||g_{k+1} - g_k|| < \epsilon$ for some tolerance ϵ

computations of the Neumann participant have to be delayed until the Dirichlet participant provides the necessary boundary conditions and vice versa.

One thing to note is that the outline provided by Algorithm 2 as is, is not well suited for time-dependent problems as the boundary conditions are synchronized only at the start and end time of the participants' computations. If the conditions changed a lot within that simulation time, this simple scheme would not contain such information. Additionally, multi-step schemes requiring intermediate values of the boundary conditions would not be able to use this scheme. This thought leads to the Dirichlet-Neumann waveform relaxation [12], that is also offered in the preCICE library through its time interpolation feature [29]. The waveform relaxation acts as a time-dependent extension of the simple Dirichlet-Neumann iteration and introduces some major changes to the algorithm. Assume $[0, T]$ be the full time interval of the coupled simulation. This interval can be split into $N \in \mathbb{N}$ subintervals $[t_i, t_{i+1}]$ with size $\Delta t = \frac{T}{N}$, we will call *time windows*. By applying the iterative algorithm on each of the time windows, we can control the synchronization frequency of the boundary conditions by specifying N with a desired value. We also allow each participant to perform timesteps of sizes $\delta t \leq \Delta t$ within a time window, with the restriction, that the last timestep has to yield the solution at the end of the current time window, which ensures non-interpolated results at those points. The data points provided by this *subcycling* allow for the construction of an interpolant, the *waveform*, of the boundary condition, which is now exchanged between the participants instead of the constant boundary condition. This enables each participant to sample the waveform at arbitrary points within the current time window, which can be used for solvers that rely on intermediate values in their computations.

3 Software

This thesis mostly revolves around the convergence behavior of SDC methods, when used in coupled simulations with preCICE. Therefore, in addition to the preCICE coupling library, libraries with existing implementations of the SDC and FEM methods are used. This section will give a brief overview of the software used in this thesis.

3.1 preCICE

preCICE [7] (***Pre**cise **C**ode **I**nteraction **C**oupling **E**nvironment*) is an open-source coupling library for partitioned multi-physics simulations, developed in the groups around Benjamin Uekermann and Miriam Schulte at the University of Stuttgart and the group of Hans-Joachim Bungartz at the Chair for Scientific Computing at the Technical University of Munich.

preCICE couples existing codes/solvers, capable of simulating a subpart of the complete physics involved in a simulation. These subparts could be programs working on different subdomains of a simulation, solvers for different physical forces acting on the same domain, or even different solvers on separate domains. This allows for high flexibility and the ability to keep a decent time-to-solution for complex scenarios, consisting of large domains and/or many physical phenomena. Its use cases include fluid-structure interaction, conjugate heat transfer (see Figure 3.1), and many more. To achieve this, preCICE offers convenient methods for transient equation coupling, communication, and data mapping.

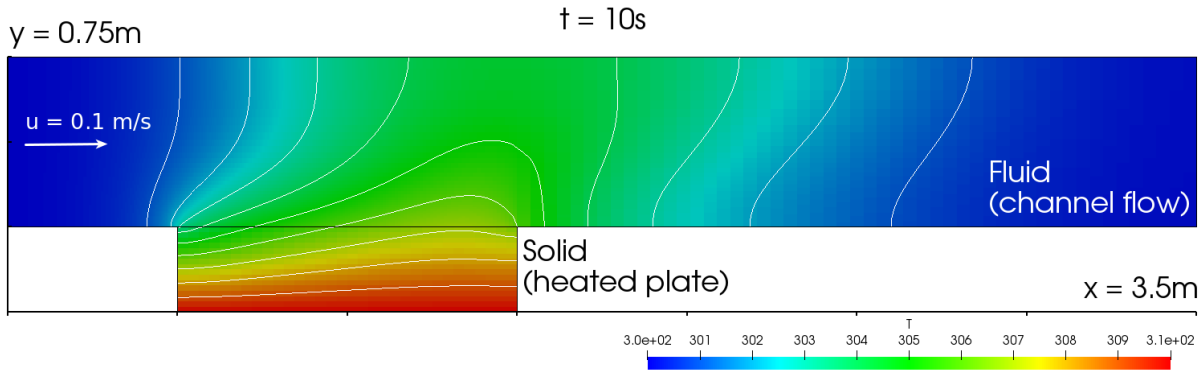

Figure 3.1 Illustration of a conjugate heat transport simulation, taken from a tutorial on the preCICE homepage[1]. The coupling of the fluid and solid solver is done with preCICE.

The core library is written in C++ but offers additional bindings for C, Fortran, Python, and Matlab. It is designed to be as minimally invasive as possible and therefore requires only a few function calls to set up a coupling between two solvers. As an entry point for new users, preCICE offers a comprehensive set of tutorials, covering several use cases and providing a step-by-step guide. For this thesis we use the official release version 3.1.2 of preCICE with the python bindings version 3.1.1.

[1] See `https://precice.org/`

3.1.1 Configuration

Since preCICE is designed to be as minimally invasive as possible, it is not directly part of the code that solves the subproblems. Therefore, necessary information about the data and the solvers has to be provided in the form of a configuration file in XML format. This file contains information about

- The form of data to be exchanged (e.g. scalars, vectors, or tensors)
- The meshes used to exchange the data (e.g. name, dimensions, data fields)
- The coupling participants and mapping scheme (e.g. name, provide and receive mesh, data fields to exchange, mapping between the meshes)
- The communication channel between the different processes (e.g. TCP/IP on the loopback network of OS)
- The coupling scheme (e.g. serial/parallel, explicit/implicit, used participants, maximum time, time-window size, data exchange directions, relative convergence measure)

3.1.2 Adapters

The preCICE ecosystem also includes an extensive list of officially maintained high-level adapters for well-known open-source solvers like OpenFOAM [8], FEniCS [28] and several more [32]. These adapters provide additional abstraction from the lower-level calls to the preCICE library, making the work with such solvers more comfortable. Apart from the officially maintained adapters, community members have also contributed adapters for other codes, which are hosted via the official preCICE Github repository[2]. In this thesis, the FEniCS-preCICE adapter is used, as FEniCS provides the spatial discretization and representation of the data.

3.1.3 FEniCS-preCICE Coupling

This subsection briefly introduces the main calls necessary to work with the FEniCS-preCICE adapter.[3]

The first step is to create an adapter instance, using its own configuration file.

```
precice = Adapter(adapter_config_filename="precice-adapter-config.xml")
```

This file contains a reference to the original preCICE configuration file, as the adapter handles the initialization of preCICE as well.

The next step is initialization, which requires information about the coupling subdomain, the function space of the read data, and a function object representing the function space of outgoing coupling data.

```
precice.initialize(coupling_boundary,
    read_function_space=V,
    write_object=flux_function)
```

Note that each participant has to be initialized according to its corresponding domain and coupling data constraints.

The data a participant receives can be requested and used via a coupling expression, which is created using

[2] A more detailed list of available adapters can be found on `https://precice.org/adapters-overview.html`

[3] We use the FEniCS-preCICE adapter version 2.1.0 in this thesis.

```
coupling_expression = precice.create_coupling_expression()
```

This expression can then be updated to retrieve the coupling data at a requested time.

```
read_data = precice.read_data(dt)
precice.update_coupling_expression(coupling_expression, read_data)
```

Similarly, coupling data is provided to preCICE in the form of a function object, which can then be read by other participants.

```
precice.write_data(function_object)
```

With these calls, simple data exchange between the solvers is possible. To create a complete simulation loop, several other calls are necessary.

- `precice.is_coupling_ongoing()` checks if `max-time`, as specified in the preCICE configuration file, has been reached. This is used to signal the end of the coupling process.
- `precice.requires_writing_checkpoint()` and `precice.requires_reading_checkpoint()` each return a boolean, indicating that preCICE requires to read or write a checkpoint of the current state to proceed with the coupling.
- `precice.store_checkpoint(data, t, time_window)` and `precice.retrieve_checkpoint()` are used to store and retrieve the checkpoint data. Such a checkpoint consists of the data provided to preCICE at a specific time for a specific time window. Time and time window are part of the checkpoint data.
- `precice.advance(dt)` is used to advance preCICE after the solver has computed one timestep. The argument `dt` is the length of the timestep the solver has performed.
- `precice.get_max_time_step_size()` returns the maximum time step size a solver is allowed to compute with. It is given by the term $\min\{t_0 + n \cdot \Delta t - t, \delta t\}$, where $t_0 + n \cdot \Delta t - t$ is the difference between the current simulation time and the end time of the current time window (characterized by the integer n) and δt the default time step size of a participant's solver. This time has to be used in subcycling to ensure that the last timestep of a solver's computation finishes at the end of the time window.

3.2 FEniCS

FEniCS [2, 20][4] is a popular open-source computing platform for solving partial differential equations (PDEs) with the finite element method (FEM)[5]. More specifically, FEniCS is an umbrella project, to provide efficient interoperability of multiple software libraries, tools, and other components.

One of the core components of FEniCS is UFL (Unified Form Language) [3]. It is a domain-specific language embedded in Python, designed to express variational forms for PDEs in a concise and readable way.

For example, the integral describing the mass matrix entries in our heat equation example

[4] Due to the fact, that currently, no finished adapter for FEniCSx (the latest iteration of FEniCS) is available, the legacy version of FEniCS is used in this thesis.

[5] More on FEniCS can be found on `https://fenicsproject.org/`

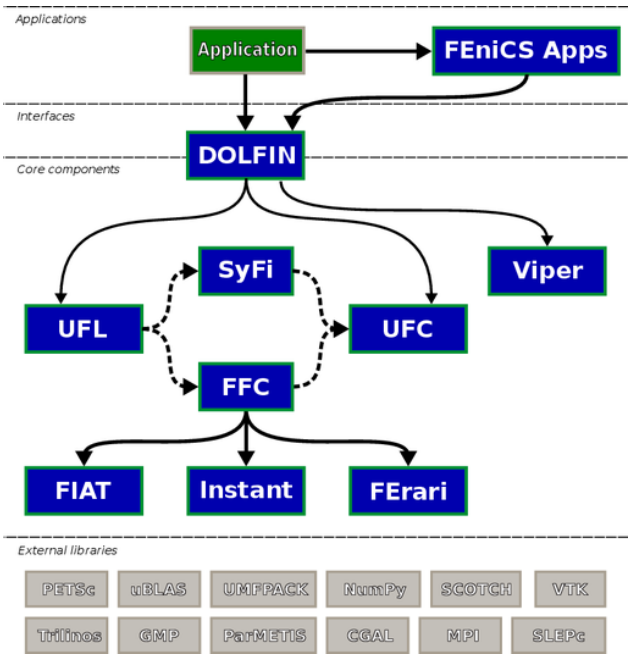

Figure 3.2 Structure of the FEniCS project. Source: `https://commons.wikimedia.org/wiki/File:Fenics-map.png`

$$\int_\Omega \phi_j \phi_i \, dx$$

can be easily expressed in UFL as

```
u * v * dx
```

where $u = \phi_j$ and $v = \phi_i$ are the trial and test functions, respectively. UFL also provides easy-to-use methods to define meshes of different shapes and sizes,

```
mesh1 = UnitSquareMesh(8, 8)
mesh2 = UnitCubeMesh(8, 8, 8)
mesh3 = RectangleMesh(Point(0, 0), Point(1, 1), 8, 8)
```

as well as function spaces suited to those meshes, for which the element family[6] and degree can be specified.

Other components of FEniCS such as FIAT (Finite Element Automated Tabulator) [14, 15], FFC (FEniCS Form Compiler) [16, 21, 26], and UFC (Unified Form-Assembly Code) are used to generate finite element basis functions, compile UFL code to UFC representation, and evaluate and assemble the variational forms efficiently.

The main interface for the user is the DOLFIN library [22, 23], which provides a C++ or a high-level Python interface to the algorithms and data structures used in FEM and Numerical Algebra. It also acts as a wrapper around the other components of FEniCS and handles communication with external libraries.

[6] There exists a plethora of different element families with different attributes. A nice overview of elements in FEM can be found in [4]

3.3 pySDC

The pySDC project [30] is a Python implementation of the spectral deferred correction (SDC) approach and its flavors, the multilevel extension MLSDC and PFASST. The simple Python interface is designed for prototyping and educational purposes, which makes it an ideal candidate to use in a thesis project.[7]

The library includes several variants of the SDC method (explicit, implicit, IMEX, multi-level, ...), some variants of the PFASST algorithm (virtual parallel for debugging, MPI-based parallel, ...) and a variety of collocation types (Gauss-Legendre, Gauss-Lobatto, ...), ready to use. All of this is delivered to the user in a heavily object-oriented fashion, which allows for easy extension and modification of the provided classes, by inheritance.

When performing a simulation, pySDC provides a high level of abstraction from the actual numerical methods used. For many use cases, it is sufficient for the user to implement only a custom problem class, choose one of the provided sweeper class and controller and provide a suitable configuration. A short introduction to those classes and the configuration and setup of a simulation is given in the following subsections.

3.3.1 pySDC Classes

As pySDC is a library designed for fast prototyping and educational purposes, there are only few components involved in the setup of a simulation. Due to the object-oriented design, they are encapsulated in classes. In this subsection we provide a short description of the classes that are the main interaction point for a user when using pySDC to setup a simulation.

The sweeper class: This class inherits from pySDC's abstract base class `sweeper` and is responsible for performing the iterative updates, the core of the SDC method. It generally contains data like solution approximations from previous steps, the time values of the integration nodes, and integration matrices. Using this information and calls to the problem class, the sweeper calculates the updated solution values for the next iteration. Like most of the classes in pySDC, the sweeper can be completely customized by the user, as long as the constraints of the base class are fulfilled. For many basic simulations, the library already provides more than ten sweepers of different types (implicit/explicit, Runge-Kutta, FEM-based, ...) and orders, which can be used out of the box.

The problem class: A problem class in pySDC should inherit from the prototype class `ptype` and is used to encapsulate problem-specific information and functionality. This includes certain parameters like physical constants and methods that are required by the sweeper. Examples of the latter are the calculation of the time derivative (`eval_f`), a method to apply the mass matrix to a given solution approximation (`apply_mass_matrix`), or a residual correction method. Which methods a user has to provide an implementation for depends on the chosen sweeper class, thus it should be selected first. By design, the problem class is the main interaction point for a user, when using pySDC to simulate a specific problem.

The controller: The heavy lifting in pySDC is done by the *controller*. It acts as a frontend endpoint for pySDC that handles the initialization of the problem and the sweeper, as well as the generation of the more involved objects pySDC uses internally, such as levels and steps. pySDC at least provides a non-MPI (runs serialized versions of the algorithms) and an MPI controller (implements parallelism; requires MPI installation). Since we are interested in convergence and error analysis, the non-MPI controller is sufficient for our purposes and allows for debugging in a serialized, deterministic fashion. These controllers can be used to execute SDC, MLSDC, and PFASST algorithms, depending on their configuration. The initialization routine for the non-MPI controller looks as follows:

[7] For the implementation in this thesis, we use a fork of the original pySDC project. The specific commit we forked from can be found at `https://github.com/Parallel-in-Time/pySDC/tree/e372a43`.

```
controller = controller_nonMPI(
                num_procs=1,
                controller_params=controller_params,
                description=description
            )
```

Regarding the arguments, we neglect the number of processes, as the non-MPI controller serializes the algorithms anyway. The more interesting parts are the dictionaries containing the controller parameters and the problem description. Such dictionaries define the behavior of the controller, the used algorithm, and the problem to solve. Therefore we will give a brief overview of the configuration.

3.3.2 Configuration Dictionaries

As mentioned in the previous subsection, the user has to provide two dictionaries as configuration for the controller, the `controller_params` and the `description` dictionary. An example of a full configuration is given below.

```
    controller_params = {
        'logger_level': logger_level
    }

    description = {
        'problem_class': fenics_heat_2d,

        # constructor arguments for the problem class
        'problem_params': {
            'function_space': function_space,
            'coupling_boundary': coupling_boundary,
            'remaining_boundary': remaining_boundary,
            'solution_expr': u_D,
            'forcing_term_expr': forcing_expr,
            'precice': precice,
            'coupling_expr': coupling_expression
        },
        'sweeper_class': imex_1st_order_mass,

        # constructor arguments for the sweeper class
        'sweeper_params': {
            'quad_type': quad_type,
            'num_nodes': num_nodes,
        },
        'level_params': {
            'restol': restol,
            'dt': dt
        },
        'step_params': {
            'maxiter': maxiter,
        }
    }
```

The `controller_params` dictionary specifies for example controller output and logging to file capabilities. The `description` dictionary is much more interesting and defines parameters regarding the simulation, the used SDC sweeper, problem parameters, and so on.

Problem and sweeper class have to be specified as entries in the `description` dictionary with the keys `'problem_class'` and `'sweeper_class'` respectively. The initialization routine of the controller then handles the instantiation of those classes. Necessary input parameters for the constructors also need

to be provided via the dictionary entries `'problem_params'` and `'sweeper_params'`, which themselves are dictionaries.

Two other major entries are `'level_params'` and `'step_params'`. A level in pySDC is a class that contains all data and functionality to perform sweeps on that particular level, including the solution, right-hand side vectors, and problem instances. The level parameters contain information, that gets forwarded to every level created in the controller. Here we can specify `'dt'` as the size of a time interval and `'restol'` as the stopping criterion when the residual after an iteration is smaller than the specified value. A collection of levels with potentially differing amounts of integration nodes or spatial grids are grouped in a hierarchy, within the step class. As the name suggests, the step class contains functionality to perform one full timestep using the contained levels. In the `step_params` dictionary we can then specify a maximum iteration limit as `'maxiters'`, to stop early in cases of slow convergence.

Since SDC without multiple levels already acts as a high-order time integration method, MLSDC is not further explored in this thesis, which leaves only few entries for our `level_params` and `step_params`. If MLSDC were to be used, one could specify different amounts of nodes and spatial subdivisions for the levels as well as transfer operators for interpolation and restriction of data between the levels. For this, pySDC allows the insertion of sweeper and problem parameters in the form of lists, where each entry serves as input for a different level. [8]

[8] For more information on how to use MLSDC with pySDC, a look at the tutorial series on `https://parallel-in-time.org/pySDC/` is suggested.

4 Implementation

This chapter describes the two implementation parts of the SDC-based solver we use for simulations, namely the problem class and the simulation/coupling loop. It has to be noted, that the current implementation only supports the use case as Dirichlet participant. For this reason, simulations in this thesis use an already existing solver from the preCICE tutorials as Neumann participant. Nevertheless, a future implementation for the Neumann case is kept in mind, by keeping the implementation easily extensible.

4.1 Problem Class

As mentioned in Subsection 3.3.1, the problem class is the central interface for the user to define the problem to be solved. Also as mentioned in the same subsection, we first decide on a sweeper class that we want to use, as it determines mandatory methods we need to implemented in the problem class. In our case, we choose the `imex_1st_order_mass` sweeper, already provided by pySDC. As the name suggests, this sweeper uses first-order time stepping as the underlying time integrator and solves one part of the problem implicitly and the other one explicitly. The ending of the name, `mass`, stems from the fact that this sweeper works with FEniCS as FEM solver and includes the mass matrix in the calculations. It has to be noted, that the SDC algorithm as described in Subsection 2.1 is a basic formulation. There are different formulations of the SDC algorithm, achieving the same goal, therefore we limited the introduction of SDC to the basic formulation. Similarly, the `imex_1st_order_mass` sweeper we use, internally uses one of those different formulations. Since we use pySDC as a tool that hides such implementation details behind a layer of abstraction, we do not need to worry about the exact formulation used. Due to the same reason, the calculation of the derivative in our implementation is split into an explicit and an implicit part, in contrast to the to the description in Section 2.2. The FEniCS example of the one-dimensional forced heat equation in the pySDC tutorials serves as a baseline for the implementation and is adapted to our needs.

With the selection of the sweeper class out of the way, we determine the methods we are required to provide. For convenience, we provide aliases for the pySDC classes, that encapsulate FEniCS meshes with additional convenience methods. By inheriting from the `ptype` class, we also get additional help through the provided abstract declarations.

Reduced to its method signatures, the final problem class looks as follows:

```
class fenics_heat_2d(ptype):
    # Aliases for the data types used for computations with FEniCS
    dtype_u = fenics_mesh       # contains a single FEniCS mesh
    dtype_f = rhs_fenics_mesh   # contains two FEniCS meshes (explicit and implicit part)

    def __init__( self, function_space, forcing_term_expr, solution_expr,
                  coupling_boundary, remaining_boundary, coupling_expr, precice ):
        ...

    def solve_system(self, rhs, factor, u0, t):
        ...

    def eval_f(self, u, t):
        ...
```

```
    def fix_residual(self, res):
        ...

    def apply_mass_matrix(self, u):
        ...
```

The usage and bodies of these methods are explained in more detail in the following subsections.

4.1.1 Constructor

The constructor is called when the controller instantiates the problem class as part of the initialization process. The constructor of the problem class used in the pySDC-FEniCS tutorial[1] handles the creation of FEniCS `Mesh`, `FunctionSpace` and `Expression` objects by itself. Its constructor arguments directly influence the properties of those objects. In our simulation, which uses many of those objects in the solver program (see Section 4.2), we create them externally and pass them into the class as constructor arguments. This way, the customization of the manufactured solution, the wanted FEniCS meshes and function space, and many more is dealt with outside of the problem class in a central location.

The constructor signature then looks as follows:

```
def __init__(   self,
                function_space,
                forcing_term_expr,
                solution_expr,
                coupling_boundary,
                remaining_boundary,
                coupling_expr,
                precice):
```

The arguments are named as self-explanatory as possible.

- `function_space` is the FEniCS function space object used for the FEM discretization.
- `forcing_term_expr` and `solution_expr` are FEniCS `Expression` objects for the forcing term and the manufactured solution, respectively.
- `coupling_boundary` and `remaining_boundary` are objects, defining the coupling boundary and the rest of the subdomain boundary in terms of the used FEM mesh. This is achieved by inheritance from the FEniCS class `SubDomain`. The implementation of the `SubDomain` member function `inside(self, x, on_boundary)` then returns a boolean, indicating if a point `x` is part of the subdomain we want to define. The argument `on_boundary` is a boolean provided by FEniCS, indicating whether the point lies on the boundary of the complete domain.
- `coupling_expr` is a FEniCS expression that is used to define the coupling boundary condition. This expression is updated each time, coupling data is requested from preCICE. It is created in the solver program using the preCICE-FEniCS adapter as mentioned in Subsection 3.1.3.
- `precice` is a reference to the preCICE-FEniCS adapter object that is used to communicate with the preCICE library. This way calls to the adapter can be performed within the problem class to update the `coupling_expr` object (see Subsection 3.1.3).

The body of the constructor can be roughly divided into three parts. The first part simply saves some of the provided arguments as class attributes for later use and invokes `__init__` on the parent class. Additionally the attribute `t_start` is defined. This attribute is used to keep track of the start time of the current time window during coupling and is updated by the simulation loop.

[1] See `https://parallel-in-time.org/pySDC/tutorial/step_7.html`

```
# Set precice reference and coupling expression reference to update coupling boundary
# at every step within pySDC
self.precice = precice
self.coupling_expression = coupling_expr
self.t_start = 0.0

# save function space for future reference
self.V = function_space

# Forcing term
self.forcing_term_expr = forcing_term_expr

# Solution expression for error comparison and as boundary condition
# on the non-coupling boundary
self.solution_expr = solution_expr

# invoke super init
super(fenics_heat_2d, self).__init__(self.V)
```

In the second part we create a trial and a test function from the specified function space which are then used to construct the mass and stiffness matrices. All of this is done via the corresponding FEniCS functions `TrialFunction(...)`, `TestFunction(...)` and `assemble(...)`.

```
# Define Trial and Test function
u = TrialFunction(self.V)
v = TestFunction(self.V)

# Mass term
a_M = u * v * dx
self.M = assemble(a_M)

# Stiffness term (Laplace)
a_K = -1.0 * inner(nabla_grad(u), nabla_grad(v)) * dx
self.K = assemble(a_K)
```

The last part handles boundary conditions. Here we set up FEniCS objects representing Dirichlet boundary conditions, by specifying the `FunctionSpace`, `Expression`, and `SubDomain` objects that define a boundary. In the case of the Dirichlet participant, the coupling boundary itself is defined as a `DirichletBC` object with the coupling expression providing the boundary values. Regardless of the type of the participant, the non-coupling boundary is always defined as a `DirichletBC` object holding the manufactured solution values. Time-dependent boundary conditions may introduce defects in the residual computation of the sweeper, thus we allow it to use a homogeneous boundary condition to fix this. For this, we create an additional `DirichletBC` object with the constant value zero on the complete domain boundary.

```
if self.precice.get_participant_name() == ProblemType.DIRICHLET.value:
    self.couplingBC = DirichletBC(self.V, coupling_expr, coupling_boundary)

self.remainingBC = DirichletBC(self.V, solution_expr, remaining_boundary)

# Allow for fixing the boundary conditions for the residual computation
# Necessary if imex-1st-order-mass sweeper is used
self.fix_bc_for_residual = True

# define the homogeneous Dirichlet boundary for residual correction
def FullBoundary(x, on_boundary):
```

```
    return on_boundary
self.homogenousBC = DirichletBC(self.V, Constant(0), FullBoundary)
```

4.1.2 Other Methods

`solve_system(self, rhs, factor, u0, t)`: This method solves the system of equations

$$(M - \text{factor} \cdot K)u = \text{rhs}$$

where M and K are the problem-specific mass and stiffness matrix respectively and factor is some scalar value.

```
def solve_system(self, rhs, factor, u0, t):
    u = self.dtype_u(u0)            # Create u as a copy of u0
    T = self.M - factor * self.K    # Create matrix T as the result of (M - factor*K)
    b = self.dtype_u(rhs)           # Create b as a copy of rhs

    # Update the solution expression to the current time and apply it as
    # Dirichlet boundary condition on the non-coupling boundary
    self.solution_expr.t = t
    self.remainingBC.apply(T, b.values.vector())

    # If the current instance is a Dirichlet participant, update the coupling expression
    # and apply it as Dirichlet boundary condition on the coupling boundary
    if self.precice.get_participant_name() == ProblemType.DIRICHLET.value:
        dt = t - self.t_start
        read_data = self.precice.read_data(dt)
        self.precice.update_coupling_expression(self.coupling_expression, read_data)
        self.couplingBC.apply(T, b.values.vector())

    # Solve the system
    solve(T, u.values.vector(), b.values.vector())

    return u
```

In this method we first create the objects u, T and b we use for the calculations later. We then update the expression for the manufactured solution to the current time t an apply it to the equation system as Dirichlet boundary condition on the non-coupling boundary.

The body of the **if**-statement is executed if the given instance of the problem class is used by a Dirichlet participant. In that case, the coupling boundary condition is given by a `DirichletBC` object as well, which is applied to the same system as before. Before application, the boundary data is requested from preCICE with the `read_data` method of the adapter object, and then used to update the coupling expression via `update_coupling_expression`. We have to keep in mind, that `read_data(dt)` expects the time difference from the start of the current time window to the time point of the requested data as an argument. For this reason, the class attribute `t_start` is necessary. It provides the class with the start time of the current time window and is kept updated by the simulation loop.

Finally, the system is solved using the FEniCS `solve` method and the solution is returned.

`eval_f(self, u, t)`: The calculation of the time derivative of the solution. Since we use an IMEX scheme, this method returns the right-hand side split up into an explicit (forcing term) and an implicit part (other terms). For this, the pySDC data type `rhs_fenics_mesh` (or better, the alias `dtype_f`) is used, which contains two FEniCS meshes, one for the explicit and one for the implicit part.

```
def eval_f(self, u, t):
    f = self.dtype_f(self.V)
```

```
    # Implicit part: Ku
    self.K.mult(u.values.vector(), f.impl.values.vector())

    # Explicit part: Mg   (g = discretized forcing term values)
    self.forcing_term_expr.t = t
    f.expl = self.dtype_u(interpolate(self.forcing_term_expr, self.V))
    f.expl = self.apply_mass_matrix(f.expl)

    return f
```

`fix_residual(self, res)`: Time-dependent boundary conditions are applied when updated solution values are computed by the sweeper. The residual computation though, is decoupled from this process and carried out without regard for the changing boundary conditions, resulting in erroneous residual values. The sweeper uses the fact, that the solution is exact at the boundary (due to the Dirichlet boundary conditions), which in turn means the residual at the boundary has to be zero. For that reason this method (`fix_residual`) is implemented to apply homogeneous Dirichlet boundary conditions (i.e. zero values at the boundary) to the residual.

```
def fix_residual(self, res):
    self.homogenousBC.apply(res.values.vector())
```

`apply_mass_matrix(self, u)`: This method is called to return the product of the mass matrix M and the input vector u. It is used by the sweeper and as a convenience method in the `eval_f` method.

```
def apply_mass_matrix(self, u):
    me = self.dtype_u(self.V)
    self.M.mult(u.values.vector(), me.values.vector())

    return me
```

4.2 Solver Program

The central piece of the simulation is the solver program. It is responsible for the setup, time-stepping, and the coupling with other solvers through the use of calls to preCICE. Here we also reuse already existing code, this time from the preCICE tutorial to the *partitioned heat equation*[2] and modify it to use pySDC and our previously created problem class as an underlying solver, instead of FEniCS alone. Large parts of the code remain unchanged as they are responsible for the setup of the participant with preCICE and the storage and logging of simulation results. Therefore, only the parts that required major changes, and the essential building blocks of the coupling process will be discussed in this section.

4.2.1 Setup

Before the solver program can start the simulation loop, there are several setup steps to be performed beforehand.

[2] See `https://github.com/precice/tutorials/tree/develop/partitioned-heat-conduction` for the full tutorial code. The distinct file, which was used as a baseline for this implementation was `https://github.com/precice/tutorials/blob/develop/partitioned-heat-conduction/solver-fenics/heat.py`

We begin with the initialization of the domain, the used function spaces and the creation of `Expression` objects for the manufactured solution (which is used as Dirichlet BC) and the forcing term. Depending on the participant type (`Dirichlet` or `Neumann`) we determine the subdomain, coupling boundary (boundary line at $x = 1$) and remaining boundary. We note again, that with the current implementation, the Neumann case is not supported, as the problem class is missing necessary modifications[3].

The creation of the `Expression` objects is done using `sympy`. We specify the formula for our manufactured solution and use the `sympy` function `diff` to create the forcing term.

```
# Define used function variables as sympy symbols
x_sp, y_sp, t_sp = sp.symbols(['x[0]', 'x[1]', 't'])

# Define manufactured solution as sympy formula and create Expression object
u_D_sp = 1 + x_sp * x_sp + alpha * y_sp * y_sp + beta * (t_sp ** temporal_degree)
u_D = Expression(sp.ccode(u_D_sp), degree=2, alpha=alpha, beta=beta, t=0)

# Define forcing term via derivatives of manufactured solution and create Expression object
f_sp = u_D_sp.diff(t_sp) - u_D_sp.diff(x_sp).diff(x_sp) - u_D_sp.diff(y_sp).diff(y_sp)
forcing_expr = Expression(sp.ccode(f_sp), degree=2, alpha=alpha, beta=beta, t=0)
```

After the definition of `sympy` symbols for the spatial coordinates and time, we define the manufactured solution as a formula and create a FEniCS `Expression` object from it. The variables `alpha`, `beta` and `temporal_degree` are used to control characteristics of the solution and are specified beforehand. Note that the formula used here is just one example and can be changed to any desired solution, depending on the use case.

The next interesting part is the initialization of the preCICE-FEniCS adapter and the coupling expression, which follows the steps described in Section 3.1.3.

```
# Create preCICE-FEniCS adapter object
precice = Adapter(adapter_config_filename="precice-adapter-config.json")

# Initialize preCICE adapter with coupling boundary and read function space and write object
precice.initialize(coupling_boundary, read_function_space=V, write_object=f_N_function)

# Create coupling expression object
coupling_expression = precice.create_coupling_expression()
```

`coupling_boundary` is an object inheriting from FEniCS' `SubDomain` class and specifies points on the boundary where coupling is performed. `read_function_space` in the case of the Dirichlet participant is just the function space for the given FEM mesh on the domain. The `write_object` is a FEniCS `Function` object on the vector function space for the flux in the direction of the Neumann participants subdomain. For the Neumann participant, the roles of the two function spaces are reversed.

The next major code part is the setup of the configuration dictionaries and the pySDC controller. This is done very similarly as described in Subsection 3.3.2.

```
# initialize controller parameters
controller_params = {
    'logger_level': logger_level
}

# fill description dictionary for easy instantiation
description = {
```

[3] Even though the Neumann case is not supported within the problem class, steps like the construction of the subdomain and the reading and writing of data to preCICE for the Neumann participant are already part of the code.

```
    'problem_class': fenics_heat_2d,
    'problem_params': {
        'function_space': function_space,
        'coupling_boundary': coupling_boundary,
        'remaining_boundary': remaining_boundary,
        'solution_expr': u_D,
        'forcing_term_expr': forcing_expr,
        'precice': precice,
        'coupling_expr': coupling_expression
    },
    'sweeper_class': imex_1st_order_mass,
    'sweeper_params': {
        'quad_type': quad_type,
        'num_nodes': num_nodes,
    },
    'level_params': {
        'restol': restol,
        'dt': dt
    },
    'step_params': {
        'maxiter': maxiter,
    }
}

# Controller for time stepping
controller = controller_nonMPI( num_procs=1,
                                controller_params=controller_params,
                                description=description )

# Reference to problem class for easy access to exact solution
P = controller.MS[0].levels[0].prob
```

Here we set `level_params[dt]` to some fixed value we require for our testing purposes. The same is true for the residual tolerance, and the type and number of nodes for the quadrature rule. Since the SDC time step size is fixed once the controller is initialized, we have to set it to some integer fraction of the time window size. Otherwise, the last step within a time window could potentially exceed the time window, which can lead to errors when requesting coupling data.

The problem parameters are set to contain all the objects the problem class requires (see Section 4.1).

The problem description dictionary is then populated with the problem class, the problem parameters, the sweeper class and parameters, as well as level and step parameters.

Finally, we initialize the controller with our configuration and create a reference to the problem class to update the variable `t_start` to the starting value of the current time window during the coupling loop.

This setup procedure is abstracted into a method, which is provided with the required parameters for `function_space`, `coupling_boundary` `dt` and so on. The method then returns the initialized controller object and the problem class reference.

4.2.2 Simulation Loop

After the setup is done, the program enters the main simulation loop, where the time-stepping and coupling with preCICE is performed by using the methods described in Section 3.1.3. An outline of the most important steps of a simulation/coupling loop is shown in Algorithm 3.

Despite the simplicity, there is quite a lot happening behind the scenes. Therefore, we will provide short explanations and code snippets for each step.

The loop itself follows the intuitive structure of repeatedly advancing the simulation until the end time is reached. Since the end time is specified in the preCICE configuration file, a call to the adapter method `precice.is_coupling_ongoing()` is used as the loop condition. After the loop is finished, a

Algorithm 3 Simulation Loop

```
while End time not reached do
    if Writing checkpoint required then
        Write checkpoint
    (1) Calculate maximum time step size
    (2) Read boundary data from preCICE
    (3) Perform a time step with the calculated length
    (4) Write resulting boundary data to preCICE
    (5) Tell preCICE to advance coupling
    if Loading checkpoint required then
        Load checkpoint
    else
        Update solution at the end of the time window
        Update time and iteration counter
Finalize the simulation
```

call to `precice.finalize()` closes communication channels and frees the other resources used by preCICE.

The two conditionals in the loop are used to handle the checkpointing mechanism of preCICE. This is necessary when using implicit coupling schemes, as their central mechanism is to move backwards in time. For this the first iteration of each time window stores a checkpoint, which can be loaded in every iteration the coupling does not converge.

```
if precice.requires_writing_checkpoint():
    precice.store_checkpoint(u_n, t, n)

...

if precice.requires_reading_checkpoint():
    u_cp, t_cp, n_cp = precice.retrieve_checkpoint()
    u_n.assign(u_cp)
    t = t_cp
    n = n_cp
```

As seen in the code snippet, the checkpointing mechanism is mostly handled by preCICE, as the function calls `requires_writing_checkpoint()` and `requires_reading_checkpoint()` signal the necessary action. Similarly, the function calls `retrieve_checkpoint()` and `store_checkpoint()` are then used to effectively read and write checkpoint data, consisting of the current solution `u_n`, time `t` and time window (iteration number) `n`.

The major part of the loop happens in the steps (1) to (5).

Step (1): preCICE demands that the final time step performed within a time window ends exactly at the time window end time. Therefore, the maximum step size is limited by the difference between the current simulation time and the end time of the current time window. This value can be requested by calling `precice.get_max_time_step_size()` and should be used to limit the time step size before performing the step. As mentioned in the previous subsection, we initialize the pySDC time step as some integer fraction of the time window size. Since we use pySDC to perform high-order time-stepping, it is generally desirable to step through a full time window with a single step, as due to the high order we still expect accurate results. In that case, the pySDC time step size is set equal to the time window size. If higher-order waveforms are used for coupling, subcycling is required to fully use the higher waveform degree. With the current implementation, one would then simply set the pySDC time step size to an integer fraction of the time window size.

Step (2): The boundary data is read and written to the coupling expression via the adapter methods `precice.read_data(dt)` and `precice.update_coupling_expression()`. In our case, this sampling of the coupling waveform is done within the problem class, as pySDC requires the coupling data to be available at several time points, depending on the type and number of quadrature nodes.

Step (3): In this step we simply perform the time stepping by calling `controller.run(...)`

```
P.t_start = t
uend, _ = controller.run(u_n, t0=t, Tend=t + float(dt))
u_np1 = uend.values
```

The argument `u_n` is the current solution, `t` is the current time, and `dt` is the size of the time step. As mentioned in Section 4.1 we provide the problem class with the start time of the time window using `P.t_start = t`, so that boundary data can be requested correctly. We then update the current solution `u_np1` with the result of the time step.

Step (4): After the time step is performed, the new boundary data for the other participant has to be calculated and written to preCICE. If the current participant is a Neumann participant, the current solution can be directly written as Dirichlet boundary data. In the case of a Dirichlet participant, the transfered data has to be the flux in the direction of the Neumann subdomain.

```
if problem is ProblemType.DIRICHLET:
    # Dirichlet problem reads temperature and writes flux on boundary to Neumann problem
    determine_gradient(V_g, u_np1, flux)
    flux_x = interpolate(flux.sub(0), W)
    precice.write_data(flux_x)
elif problem is ProblemType.NEUMANN:
    # Neumann problem reads flux and writes temperature on boundary to Dirichlet problem
    precice.write_data(u_np1)
```

Step (5): After performing a timestep we need to tell preCICE, how far we proceeded such that the coupling can be advanced correctly. As many things with the preCICE-FEniCS adapter, this results in a single function call `precice.advance(dt)`, where `dt` is the size of the time step we just performed.

5 Methodology and Results

5.1 Method of Manufactured Solutions

The method of manufactured solutions (MMS) provides a convenient way to create a PDE with a known analytical solution. At its core, the method consists of the selection of a suitable, non-trivial, analytical solution to a given problem and the following derivation of an additional forcing term that is added to the original problem.

We can then supply the modified problem we created using the MMS to our numerical solver and compare the computed solution to the known manufactured solution. This way we can verify the correctness of the implementation and the convergence behavior of our numerical method. The verification process is carried out by systematically monitoring the convergence of some error measure with decreasing time step size (δt; used for the monolithic solvers without coupling) or time window size (Δt; used for all coupled examples). While there are various error measurements, they all depend on the difference between the approximated and the manufactured analytical solution.

For our testing purposes, we apply the MMS to our test problem, the forced heat equation. Since the heat equation is a somewhat common showcase problem in the context of the used software tools [1], we will use a modified version of the MMS example used in the FEniCS tutorial book [18], where the time term is raised to some power n.

$$
\begin{aligned}
&u = 1 + x^2 + \alpha y^2 + \beta t^n \\
&\alpha = 3.0 \\
&\beta = 1.2 \\
&n \ldots \text{Exponent of the time term}
\end{aligned}
$$

By adjusting this exponent, we can investigate the convergence behavior for cases with non-linear behavior in time.

Starting with the original heat equation, we can now insert the manufactured solution to derive the forcing term f, that characterizes the development of the system over time.

$$\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f$$

$$\frac{\partial u}{\partial t} = \beta n t^{n-1} \qquad \frac{\partial^2 u}{\partial x^2} = 2 \qquad \frac{\partial^2 u}{\partial y^2} = 2\alpha$$

$$f = \beta n t^{n-1} - 2 - 2\alpha$$

Observing the solution term above, one can find several issues that can influence coupling results. For one, the flux in the direction of the Neumann participant (in our case increasing x direction) is constant over time. Also, for small n, the participants should perform exact time integration if the order of the time stepping scheme is higher than that of the time term in the solution. We use an additional manufactured solution to test cases where these issues are absent.

$$u = 1 + (1 + \sin(t)) \cdot x^2 + \alpha y^2 + \beta t$$

We derive the forcing term for this solution in the same way as before.

[1] FEniCS, pySDC, and preCICE provide a tutorial using the heat equation as an example

The addition of the time-dependent factor also makes the flux in x-direction time-dependent. Also, the sine-function is not a polynomial function. For this reason, the time-dependent behavior of the solution can only be approximated, regardless of the time stepping scheme.

Another critical aspect of both manufactured solutions is that the spatial degree in each dimension is quadratic. This way, we can choose the degree of the elements in the FEM mesh to be the same, which should result in no additional error due to spatial discretization.

5.2 Results

With the manufactured solutions and the corresponding forcing terms, we test the pySDC implementation with a monolithic simulation. This way, we can check if the method reproduces the analytical solution up to the expected order of convergence. Afterward, we investigate the results of simulations in a coupled setting with preCICE.

As used in previous chapters, we conduct simulations on a complete domain of $[0, 2] \times [0, 1]$ on a time interval of $[0, 1]$ and a domain decomposition along the line $x = 1$ for coupled simulations (as seen in Figure 2.1). Regarding the pySDC configuration, we use the quadrature type `'LOBATTO'`, such that the integration bounds are part of the nodes and the corresponding function values don't have to be interpolated.

5.2.1 Monolithic Simulations

First, we investigate the convergence behavior of the pySDC implementation in a monolithic setting. This way, we can verify the correctness of the implementation and the expected order of convergence. The L_2 error at the end of the simulation is used as error measurement. From [6], we know that the maximum achievable convergence order with Gauss-Lobatto quadrature nodes is $\mathcal{O}(\delta t^{2M-2})$, where M is the number of nodes, though this convergence is only expected for sufficiently small time steps and sufficient iterations. If that is the case, each iteration increases the formal order of the method by one (for implicit/explicit Euler as a low-order scheme) up to the convergence order of the underlying quadrature rule. To allow pySDC to achieve good convergence for the given time step sizes, we set the residual tolerance to 10^{-13}, and allow for a maximum of 40 iterations during monolithic simulations.

Time step sizes are decreased from $\delta t = 1$ by a factor of 2 for each simulation, yielding the test sizes

$$\delta t \in \{2^{-i} \mid i = 0, \dots, 6\}$$

For the first series of simulations, we use the manufactured solution

$$u = 1 + x^2 + 3.0y^2 + 1.2t^{64}$$

and several settings for the number of quadrature nodes M. The temporal exponent 64 allows us to investigate the convergence behavior for a higher number of nodes. If it were too small, the simulation would be already very accurate for large time step sizes and thus provide no meaningful information about the convergence behavior.

Table 5.1 shows the resulting data with two columns for every M. The first one contains the L_2 error for the given M with different values for δt. The second one contains the order of the error decrease compared to the previous δt. This error decrease can achieve a theoretical maximal convergence order of $2M - 2$ for M nodes. From the second column, it is easy to see that the error reduction does not show this maximal order but instead approaches it with decreasing δt. Additionally, for $M = 16$, the error is reduced to machine precision for $\delta t \leq 0.125$. The convergence behavior is visualized in Figure 5.1, with additional lines for the maximal theoretical convergence order. The improvement in convergence rate with decreasing δt is visible as a curvature in the plotted graphs. Even though the order $2M - 2$ is not reached for our smallest values of δt, it follows the expected behavior and shows adequate convergence for the high temporal exponent of 64 in the manufactured solution.

δt	$M = 2$		$M = 4$		$M = 8$		$M = 16$	
	L_2 error	Order	L_2 error	Order	L_2 error	Order	L_2 error	Order
1	1.56e+00		9.95e-01		1.56e-01		6.98e-04	
0.5	1.32e+00	0.23	5.70e-01	0.80	1.78e-02	3.13	6.20e-06	6.82
0.25	9.97e-01	0.41	2.01e-01	1.50	1.01e-03	4.14	4.87e-09	10.31
0.125	6.19e-01	0.69	3.17e-02	2.67	1.96e-05	5.69	9.62e-14	15.63
0.0625	2.83e-01	1.13	2.39e-03	3.73	1.35e-07	7.18	8.54e-14	*
0.03125	9.24e-02	1.61	1.18e-04	4.35	4.35e-10	8.28	7.38e-14	*
0.015625	2.51e-02	1.88	4.34e-06	4.76	1.83e-13	11.21	3.17e-14	*

Table 5.1 L_2 error for monolithic simulations with the manufactured solution $u = 1 + x^2 + 3.0y^2 + 1.2t^{64}$ and a varying number of quadrature nodes M. The second column for each node count shows the order of the error reduction compared to the previous δt. For 16 nodes, the error is approximately reduced to machine precision for $\delta t \leq 0.125$, therefore, the error reduction after that point provides no meaningful information (marked with *).

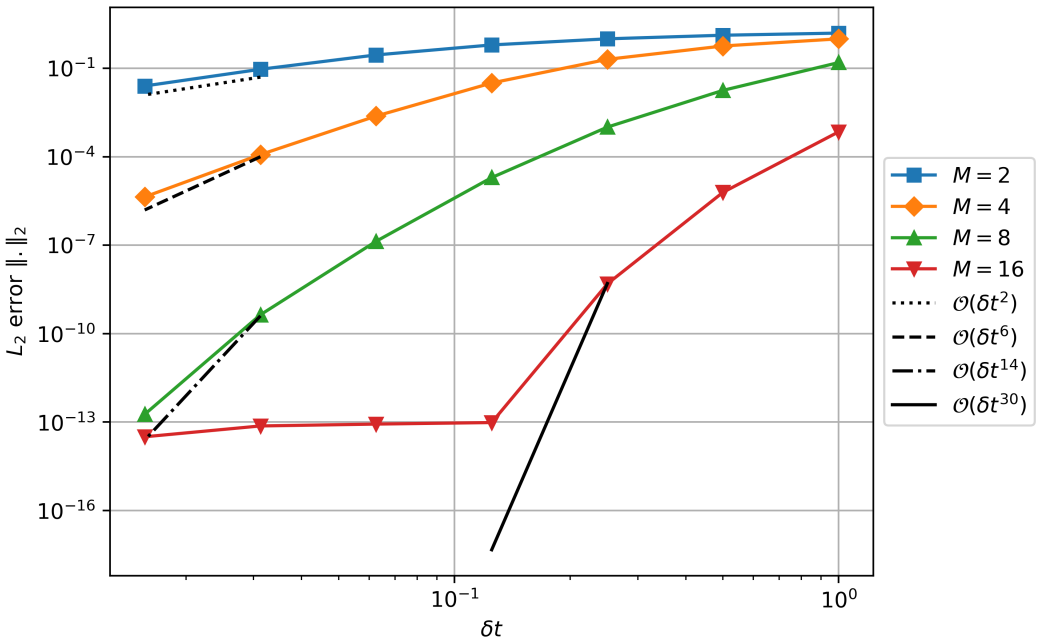

Figure 5.1 L_2 error plot for monolithic simulations with the manufactured solution $u = 1+x^2+3.0y^2+1.2t^{64}$ and varying number of quadrature nodes M. The additional lines show the maximal theoretical convergence order of $2M - 2$ for the given number of nodes. The data for this plot is given in Table 5.1.

For the following example, we lower the temporal exponent to $n = 16$.

The results (see Table 5.2 and Figure 5.2) show, a faster overall convergence rate, and a faster approach to the maximal theoretical convergence order of $2M - 2$. Additionally, we see that for solution polynomials with temporal order $n \leq M$ (in this case, $n = M = 16$), the computed solution is already accurate up to machine precision with a single large timestep.

δt	$M = 2$		$M = 4$		$M = 8$		$M = 16$	
	L_2 error	Order	L_2 error	Order	L_2 error	Order	L_2 error	Order
1	3.51e-01		1.45e-01		8.99e-04		8.63e-14	
0.5	2.65e-01	0.41	3.37e-02	2.10	1.03e-05	6.45	9.64e-14	*
0.25	1.47e-01	0.85	2.70e-03	3.64	4.40e-08	7.87	9.72e-14	*
0.125	5.30e-02	1.47	1.13e-04	4.58	1.08e-10	8.67	9.54e-14	*
0.0625	1.48e-02	1.84	3.92e-06	4.85	1.11e-13	9.92	9.51e-14	*
0.03125	3.81e-03	1.96	1.19e-07	5.04	9.57e-14	*	9.47e-14	*
0.015625	9.61e-04	1.99	3.35e-09	5.15	9.42e-14	*	9.40e-14	*

Table 5.2 L_2 error and error reductions for monolithic simulations with the manufactured solution $u = 1 + x^2 + 3.0y^2 + 1.2t^{16}$ and a varying number of quadrature nodes M. Cells marked with an asterisk (*) indicate that the error is reduced to machine precision, and a the order of the error reduction would provide no meaningful information.

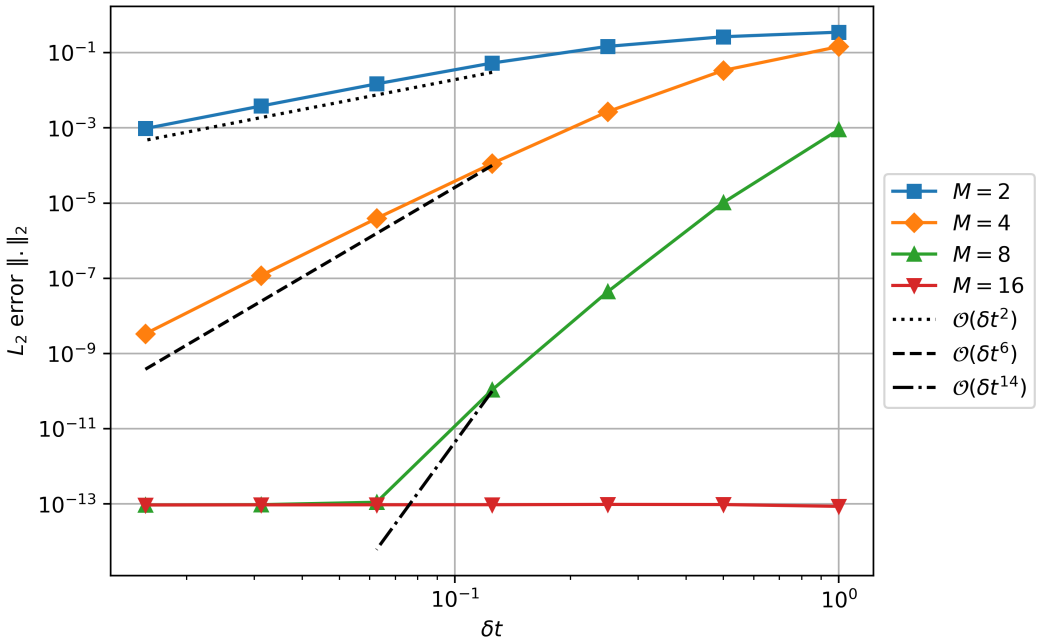

Figure 5.2 L_2 error plot for monolithic simulations with the manufactured solution $u = 1 + x^2 + 3.0y^2 + 1.2t^{16}$ and a varying number of quadrature nodes M. The additional lines show the maximal theoretical convergence order of $2M - 2$ where applicable. The data for this plot is given in Table 5.2.

Simulations with our second manufactured solution $u = 1 + (1 + \sin(t)) \cdot x^2 + 3.0y^2 + 1.2t$ show similar behavior for the smaller number of nodes as the previous polynomial examples. For $M = 8$ and $M = 16$ though, the convergence rate reaches only a order of around ≈ 2.5 in both cases. It also does not seem to approach the theoretical maximum order of $2M - 2$ as in the polynomial example. This can be seen in Table 5.3 and Figure 5.3.

An interesting observation is that the error for $M = 8$ is lower than for $M = 16$ for $\delta t \geq 0.0625$.

Another major difference to the polynomial solution is a gradual reduction in convergence order when approaching errors comparable to machine precision. This poses a stark contrast to the clean cut, present in the data for the polynomial solution. An explanation for this could be the accumulation of the high-order error terms over the course of the simulation.

δt	$M = 2$		$M = 4$		$M = 8$		$M = 16$	
	L_2 error	Order	L_2 error	Order	L_2 error	Order	L_2 error	Order
1	2.60e-03		1.27e-05		3.11e-09		1.90e-08	
0.5	1.65e-04	3.97	3.15e-07	5.34	1.23e-10	4.66	2.04e-09	3.22
0.25	6.69e-05	1.31	7.42e-09	5.41	2.45e-11	2.33	3.24e-10	2.65
0.125	1.67e-05	2.00	1.74e-10	5.41	5.00e-12	2.29	5.57e-11	2.54
0.0625	4.17e-06	2.00	4.51e-12	5.27	7.91e-13	2.66	6.73e-12	3.05
0.03125	1.04e-06	2.00	6.09e-13	2.89	1.40e-12	*	5.36e-13	3.65
0.015625	2.60e-07	2.00	3.01e-13	1.02	1.63e-12	*	3.06e-13	0.81

Table 5.3 L_2 error and error reductions for monolithic simulations with the manufactured solution $u = 1 + (1 + \sin(t)) \cdot x^2 + 3.0y^2 + 1.2t$ and a varying number of quadrature nodes M. Cells marked with an asterisk (*) indicate that the error is reduced to some lower bound, and the order of the error reduction would provide no meaningful information.

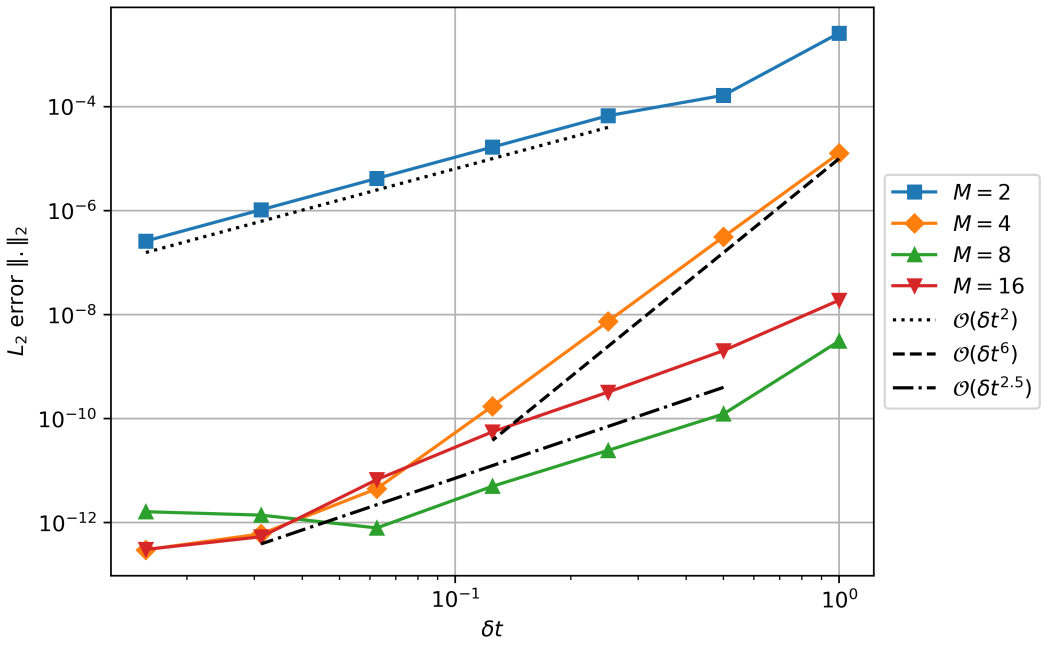

Figure 5.3 L_2 error plot for monolithic simulations with the manufactured solution $u = 1 + (1 + \sin(t)) \cdot x^2 + 3.0y^2 + 1.2t$ for a varying number of nodes M. For $M = 2$ and $M = 4$ the reference lines show the theoretical maximum convergence order of $2M - 2$. Since $M = 8$ and $M = 16$ deviate strongly from this expected behavior, the reference line has not the same meaning and is only used to provide a visual reference of the actual behavior.

5.2.2 Coupled Simulations

After the simulations in a monolithic setting, we investigate our SDC solver's behavior in a coupled setting. For this, we partition the domain along the line $x = 1$, as mentioned at the beginning of this chapter. We only use our SDC-based solver for the Dirichlet participant, while the Neumann participant uses the Gauss-Legendre method, a family of higher-order implicit Runge-Kutta schemes. The implementation for the latter is part of the preCICE tutorials and was developed and analyzed as part of Niklas Vinnitchenko's bachelor thesis [33].

Regarding the configuration file for preCICE, the existing configuration for the partitioned heat conduction tutorial is used.[2] It is mainly left unchanged, and only values regarding the maximum iterations, time window size, waveform degree, relaxation constant, and relative convergence limit are adjusted. Some tests showed that the maximum iterations can be left at 100 as the coupling converged within 40 or fewer iterations. For similar reasons, the relaxation constant is set to 0.5. The relative convergence limit is fixed to 10^{-11} as Niklas Vinnitchenko also provided simulations showing that lower settings than this only offer marginal improvements.

The waveforms preCICE uses for coupling are of a predefined degree. With the current version of preCICE[3], the maximum waveform degree is 3, with the developer version already allowing for higher degrees. Since we use the official release, we limit the number of nodes to 4 for the SDC participant and use a Gauss-Legendre method with 3 stages for the Neumann participant. This way, both solvers are of maximum order 6, which should make the coupling scheme with its waveform degrees lower than 4 the limiting factor for accuracy.

In the coupled simulations, the error is observed for decreasing time window size Δt. The time step sizes of the solvers are then adjusted to provide the least amount of data points within a time window for the used waveform degree.

From here on, we use the domain partitioning specified at the beginning of this subsection and the following settings if not mentioned otherwise.

Settings: preCICE and solvers

- Simulation time: $t \in [0, 1]$
- Time window sizes: $\Delta t \in \{2^{-i} \mid i = 0, \dots, 6\}$
- pySDC used for Dirichlet participant, FEniCS-based Gauss-Legendre method used for Neumann participant
- Solver time step sizes: $\delta t_{\mathcal{D}} = \delta t_{\mathcal{N}} = \frac{\Delta t}{p}$ with p as the waveform degree

Settings: pySDC

- `num_nodes = 4`
- `res_tol = 1e-11`
- `max_iters = 40`

We conduct the first coupled simulations with the manufactured solution

$$u = 1 + x^2 + 3.0y^2 + 1.2t^2$$

and the waveform degrees $p \in \{1, 2, 3\}$.

The error in both participants' domains is measured at the end of the simulation, but by comparing the behavior of the errors, one can see very high similarity, both in convergence order and error magnitude, as depicted in Figure 5.4. The same is true for all conducted simulations, so from here on, we only refer to the error of the Dirichlet participant.

[2] We used the preCICE tutorials v202404.0. The configuration file can be found at `https://github.com/precice/tutorials/tree/v202404.0/partitioned-heat-conduction`

[3] We used the preCICE release 3.1.2, together with the python bindings `pyprecice` 3.1.1

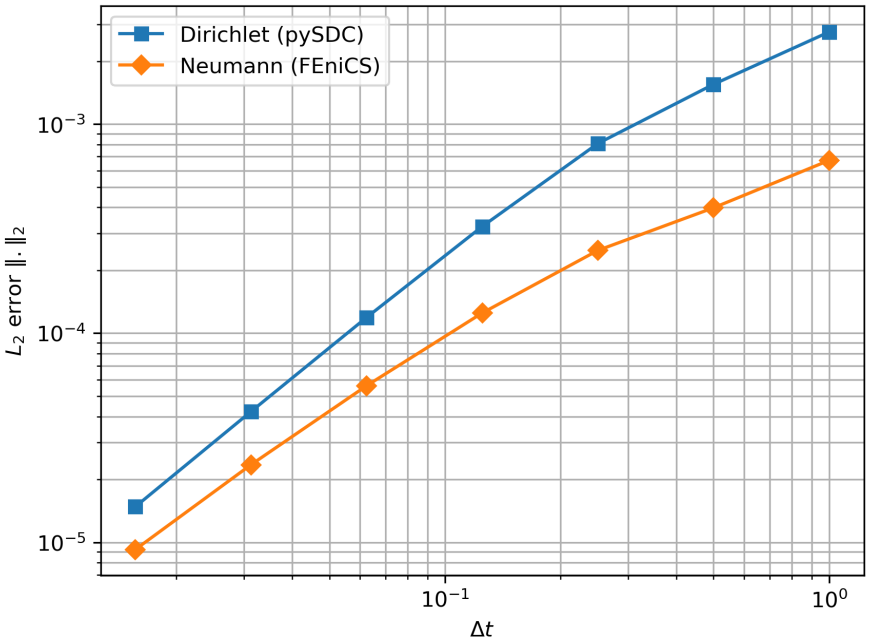

Figure 5.4 L_2 error on each domain of the coupled simulation with manufactured solution $u = 1 + x^2 + 3.0y^2 + 1.2t^2$ and waveform degree $p = 1$. The errors behave very similarly for both participants and are not too different in magnitude. Similar behavior is observed for all conducted simulations, thus only the error of the Dirichlet participant is considered for other plots and observations.

When comparing the errors between the coupled simulation with different waveform degrees and the monolithic simulation (depicted in Table 5.4 and Figure 5.5), it becomes clear that the limiting factor for the accuracy of the simulation is the coupling scheme. The first observation we make is that the monolithic simulation, as well as the coupled simulations with waveform degree $p = 2$ and $p = 3$ yield almost no error for all time step sizes. The remaining error is unavoidable due to limited machine precision. From Subsection 5.2.1, these results are expected for the monolithic simulation since the temporal degree of the solution is lower than the number of nodes used in pySDC. The behavior for the coupled cases stems from the fact that a used waveform degree p allows for exact boundary conditions up to the same degree, with introduced errors of order $\mathcal{O}(\Delta t^{p+1})$. Since for this manufactured solution, $p = 2$ and $p = 3$ provide exact boundary conditions to both participants, the corresponding simulations also perform exact time integration. Another interesting fact is that the unavoidable error for those simulations slightly increases with decreasing time window size. This can be attributed to machine precision errors accumulating for each time window.

For $p = 1$, the coupling scheme provides only linear interpolation of the boundary conditions within a time window. Therefore, we would expect the introduction of an error of order $\mathcal{O}(\Delta t^2)$. Instead, we observe a convergence order of around $\mathcal{O}(\Delta t^{1.5})$.

Δt	Monolithic		Coupled $p = 1$		Coupled $p = 2$		Coupled $p = 3$	
	L_2 error	Order	L_2 error	Order	L_2 error	Order	L_2 error	Order
1	8.58e-14		0.00276		6.39e-14		7.59e-14	
0.5	1.02e-13	*	0.00155	0.83	7.89e-14	*	5.78e-14	*
0.25	1.22e-13	*	0.000808	0.94	6.02e-14	*	7.12e-14	*
0.125	9.62e-14	*	0.000324	1.32	7.40e-14	*	7.87e-14	*
0.0625	1.07e-13	*	0.000119	1.45	8.40e-14	*	9.41e-14	*
0.03125	1.08e-13	*	4.22e-05	1.50	1.07e-13	*	1.31e-13	*
0.015625	1.07e-13	*	1.48e-05	1.51	1.55e-13	*	2.08e-13	*

Table 5.4 L_2 error for the manufactured solution $u = 1 + x^2 + 3.0y^2 + 1.2t^2$. The error is given for the full domain in a monolithic simulation and the Dirichlet participant's domain in coupled simulations with waveform degrees $p \in \{1, 2, 3\}$. The convergence from one time window size to the next is given in the second column for each simulation. Cells marked with an asterisk (*) indicate that the error is reduced to machine precision, and a reduction factor would provide no meaningful information. An interesting fact is that the error for $p = 1$ converges with a rate of at most $\mathcal{O}(\Delta t^{1.5})$ instead of the expected $\mathcal{O}(\Delta t^2)$.

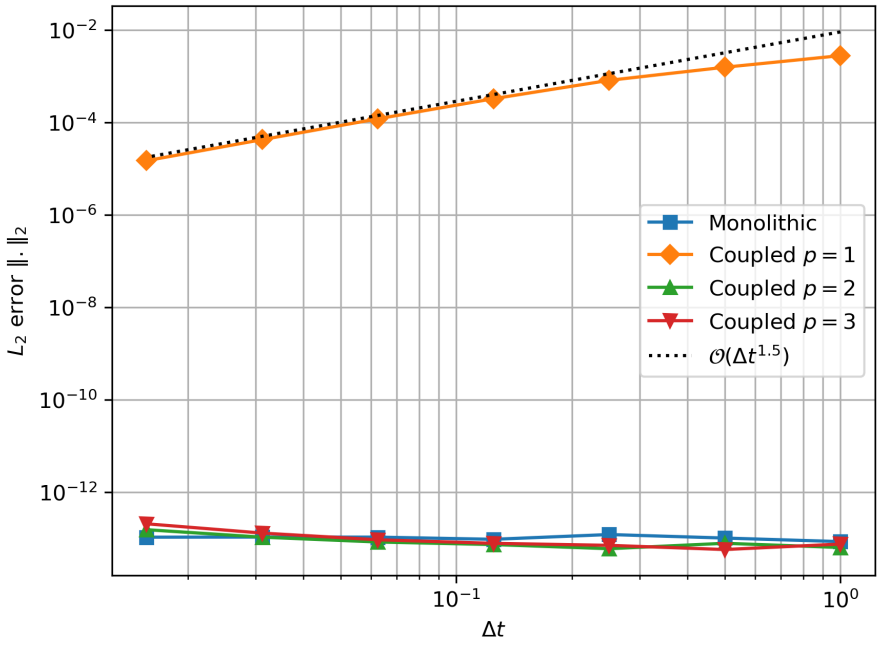

Figure 5.5 L_2 error plot for different simulations with the manufactured solution $u = 1 + x^2 + 3.0y^2 + 1.2t^2$. p denotes the waveform degree used in the coupled simulations. The data for this plot is given in Table 5.4. For the coupled simulations only the error of the Dirichlet participant is considered.

We conduct another series of simulations with the manufactured solution $u = 1+x^2+3.0y^2+1.2t^4$ to investigate the convergence behavior for the higher waveform degrees. In this case, due to the temporal degree of the solution being higher than the waveform degrees, each coupled simulation should show some converging error term. For the monolithic case, we still expect exact integration. The results are depicted in Table 5.5 and Figure 5.6.

Δt	Monolithic		Coupled $p = 1$		Coupled $p = 2$		Coupled $p = 3$	
	L_2 error	Order	L_2 error	Order	L_2 error	Order	L_2 error	Order
1	8.75e-14		0.0129		0.00486		0.000713	
0.5	9.93e-14	*	0.00763	0.76	0.00112	2.12	5.55e-05	3.68
0.25	1.19e-13	*	0.00395	0.95	0.000167	2.75	3.48e-06	4.00
0.125	9.31e-14	*	0.00165	1.26	1.94e-05	3.11	2.26e-07	3.94
0.0625	1.04e-13	*	0.000614	1.43	1.92e-06	3.34	1.60e-08	3.82
0.03125	1.04e-13	*	0.00022	1.48	1.73e-07	3.47	1.11e-09	3.85
0.015625	1.04e-13	*	7.70e-05	1.51	2.12e-08	3.03	8.61e-11	3.69

Table 5.5 L_2 error for the manufactured solution $u = 1 + x^2 + 3.0y^2 + 1.2t^4$. The error is given for the full domain in a monolithic simulation and the Dirichlet participant's domain in coupled simulations with waveform degrees $p \in \{1, 2, 3\}$. The error convergence from one time window size to the next is given in the second column for each simulation. Cells marked with an asterisk (*) indicate that the error is reduced to machine precision, and a reduction factor would provide no meaningful information.

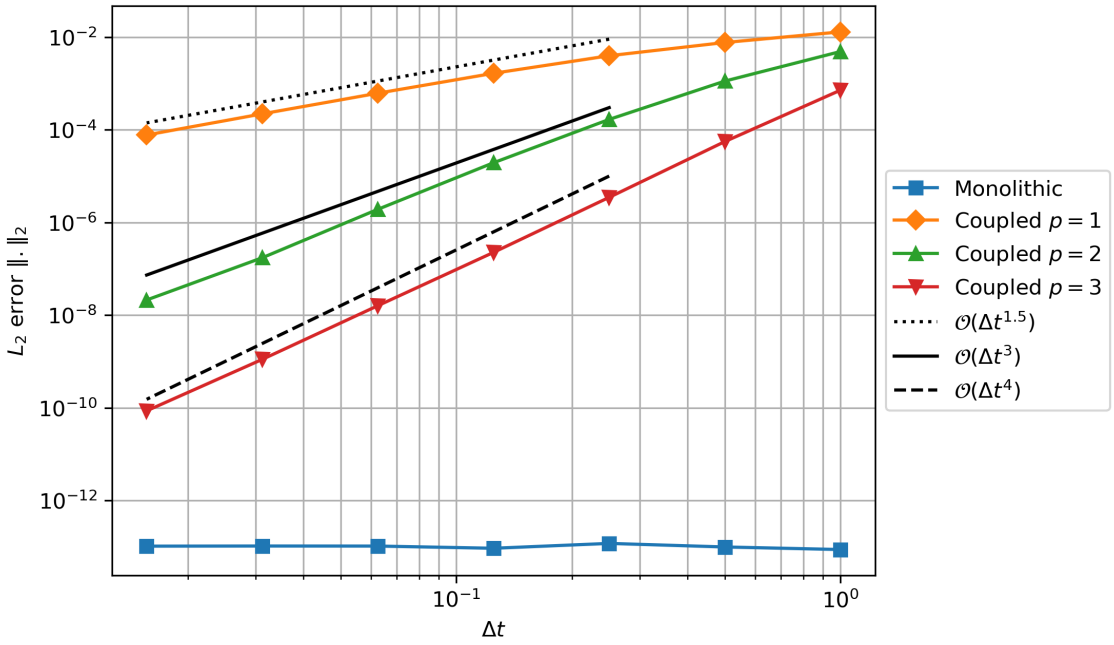

Figure 5.6 L_2 error plot for different simulations with the manufactured solution $u = 1 + x^2 + 3.0y^2 + 1.2t^4$. p denotes the waveform degree used in the coupled simulations. $p = 2$ and $p = 3$ show the expected convergence rates of $\mathcal{O}(\Delta t^3)$ and $\mathcal{O}(\Delta t^4)$ respectively. The error for $p = 1$ converges with a rate around $\mathcal{O}(\Delta t^{1.5})$ slower than the expected $\mathcal{O}(\Delta t^2)$. The monolithic solver shows no error, as the temporal order of the solution is equal to the number of nodes used in pySDC.

Again, the monolithic solution performs as expected. For the coupled simulations, this time, the convergence for each waveform degree is visible. As for the previous simulation series, the error for $p = 1$ converges with a rate of around $\mathcal{O}(\Delta t^{1.5})$, while the errors for $p = 2$ and $p = 3$ converge with approximately the expected rate of $\mathcal{O}(\Delta t^3)$ and $\mathcal{O}(\Delta t^4)$ respectively.

Another series of simulations is conducted with our second manufactured solution $u = 1 + (1 + \sin(t)) \cdot x^2 + 3.0y^2 + 1.2t$. Here, we want to look for the same behavior as in the previous simulations. This time, the flux in x-direction is time-dependent, and high-order errors are introduced due to the sine term. Those high-order errors are the reason we conduct these simulations with a higher residual tolerance for pySDC of 10^{-13}, to eliminate instabilities due to high residuals.

Δt	Monolithic		Coupled $p = 1$		Coupled $p = 2$		Coupled $p = 3$		Coupled $p = 6$	
	L_2 error	Order	L_2 error	Order	L_2 error	Order	L_2 error	Order	L_2 error	Order
1	1.27e-05		0.00068		0.000151		1.03e-05		4.59e-09	
0.5	3.15e-07	5.34	0.000355	0.94	2.48e-05	2.61	1.06e-06	3.28	2.90e-10	3.98
0.25	7.42e-09	5.41	0.000187	0.92	3.07e-06	3.01	7.33e-08	3.85	1.24e-11	4.55
0.125	1.74e-10	5.41	7.73e-05	1.27	3.21e-07	3.26	4.84e-09	3.92	6.25e-13	4.31
0.0625	4.51e-12	5.27	2.92e-05	1.40	3.04e-08	3.40	3.61e-10	3.74	1.95e-12	*
0.03125	6.09e-13	2.89	1.07e-05	1.45	2.68e-09	3.50	3.21e-11	3.49	4.30e-12	*
0.015625	3.01e-13	1.02	3.85e-06	1.47	3.64e-10	2.88	2.94e-12	3.45	3.75e-12	*

Table 5.6 L_2 error and convergence orders for different simulations with the manufactured solution $u = 1 + (1 + \sin(t)) \cdot x^2 + 3.0y^2 + 1.2t$. p denotes the waveform degree used in the coupled simulations. To prevent instabilities due to high residuals, the residual tolerance for pySDC is set to 10^{-13} during these simulations.

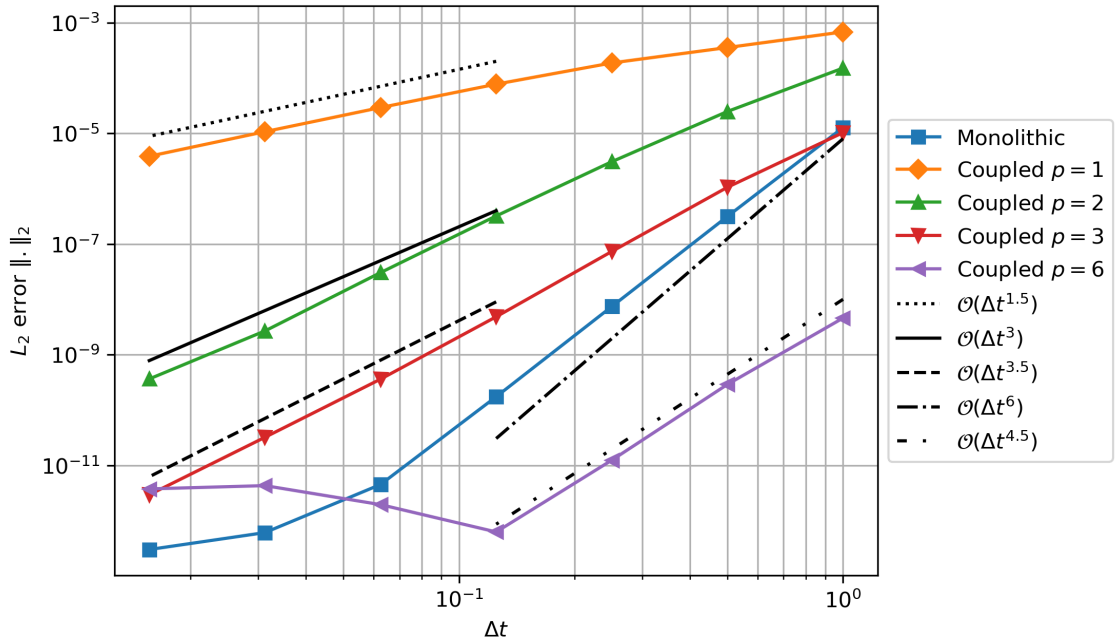

Figure 5.7 L_2 error plot for different simulations with the manufactured solution $u = 1 + (1 + \sin(t)) \cdot x^2 + 3.0y^2 + 1.2t$. p denotes the waveform degree used in the coupled simulations. In these simulations, the residual tolerance for pySDC is set to 10^{-13}, compared to 10^{-11} in previous simulations. $p = 1$ and $p = 3$ show lower convergence rates than the expected ones with $\mathcal{O}(\Delta t^{1.5})$ and $\mathcal{O}(\Delta t^{3.5})$ respectively. The monolithic simulation with no coupling is again only limited in convergence by the time integrator. The resulting convergence is as expected slightly lower than the theoretical maximum of $\mathcal{O}(\Delta t^6)$.

The results are shown in Table 5.6 and Figure 5.7. As in Subsection 5.2.1, the monolithic simulation converges for high-order solutions with slightly lower rates than the theoretical maximum. The coupled simulations show a behaviour similar to the previous simulations with the polynomial solution. The only bigger deviation is the lower convergence rate for $p = 3$ with $\approx \mathcal{O}(\Delta t^{3.5})$ instead of $\mathcal{O}(\Delta t^4)$. A similar result was already observed in [29] with waveform degree $p = 3$, an SDC-based time integrator (3 Gauss-Lobatto nodes) and a polynomial term in the manufactured solution instead of a trigonometric one. The similar behavior even for different manufactured solutions could hint towards some general issue, lowering the the convergence rate slightly when coupling an SDC-based solver.

For completion, we also conducted an additional simulation with a waveform degree of $p = 6$. For this we used the current developer version of preCICE, which allows for higher waveform degrees[4].

[4]The specific commit we used can be found at `https://github.com/precice/precice/commit/4eb1dee`

Even though the solvers provide a maximal order of 6, it seems that for high waveform degrees in the coupling schemes now the time integration is the limiting factor. If all parts were of the same order, the error would show improvements of order $\mathcal{O}(\Delta t^6)$. Instead we only observe a convergence rate of $\approx \mathcal{O}(\Delta t^{4.5})$, until unavoidable errors provide a lower bound. An explanation for this behavior could be the lowered order of the SDC solver for large time step sizes, as observed in the monolithic simulations. Though the waveform degree of 6 leads to the time step sizes of $\Delta t/6$, which should somewhat mitigate this effect. Also curious is the fact that the convergence rate stays very constant during the part of decreasing error.

Even though it is not directly part of the thesis topic we also want to point out a substantially longer runtime for the simulations with the trigonometric solution in comparison to the polynomial one. This is partially explained by the higher residual tolerance, used in the simulations with the trigonometric term. However, many of the polynomial simulations used far fewer SDC iterations than the trigonometric ones, even for a lower residual tolerance. This especially applies to small time step sizes, where some steps for the polynomial solution stopped after single-digit amounts of iterations, while the trigonometric solution required still more than 30 iterations. This behavior once more hints towards the high-order error terms in the trigonometric case causing slower convergence in the SDC solver.

6 Conclusion

6.1 Summary

In this thesis, we developed an SDC-based solver, which we used to simulate the forced heat equation in monolithic and partitioned scenarios. The obtained results provide insights into the convergence behavior of the solver and the coupling process.

The monolithic simulations with different amounts of SDC nodes showed a significant difference in convergence between solutions with polynomial and trigonometric time-dependent behavior. We also showed that using the waveform relaxation for coupling with waveform degrees lower than the solver's order leads to a lowered convergence rate. While waveform degrees equal to the solver's order still only provide a lower convergence rate than a purely monolithic simulation, the initial error is of a lower magnitude than expected.

Even if there are reductions in the convergence order for coupled simulations, the obtained convergence results show that the SDC-based time integrator is still a viable option for coupling. The created implementation gives a reasonable basis for future investigation and can be expanded upon to potentially leverage the SDC-based PinT algorithm PFASST for better computation speeds. We also contributed the code used for the simulations and results presented in this thesis to the preCICE tutorials as a pull request[1].

6.2 Future Work

For future work, the problem class and solver program used in this thesis can be expanded to support the use case as a Neumann participant. This would allow testing with only SDC-driven participants.

Another thing to investigate is the expansion of the solver to PFASST and the behavior in a highly parallelized environment. The error analysis with a single time window and high waveform degrees could be interesting in this context.

As mentioned in the last point in Chapter 5, the trigonometric term in the solution causes an increased runtime of the SDC method due to the increased number of iterations necessary for convergence. An interesting point for future work in these regards would be to adaptively change the maximum iterations or the residual tolerance of the SDC method based on the convergence state of the coupling scheme. An idea would be to increase the maximal number of SDC iterations as the coupling scheme converges. This way, the runtime could be reduced by not wasting iterations on the somewhat inaccurate boundary conditions at the start of the coupling process.

[1] The pull request can be found at `https://github.com/precice/tutorials/pull/557`

Abbreviations

ODE ordinary differential equation

PDE partial differential equation

SDC spectral deferred correction

PinT parallel-in-time

PFASST Parallel Full Approximation Scheme in Space and Time

MGRIT Multigrid Reduction in Time

FEM finite element method

MMS method of manufactured solutions

MLSDC Multi-Level Spectral Deferred Correction

BC boundary condition

Bibliography

[1] Mohamad Abidin and Md Yushalify Misro. Numerical simulation of heat transfer using finite element method. *Journal of Advanced Research in Fluid Mechanics and Thermal Sciences*, 92:104–115, 03 2022.

[2] Martin S. Alnaes, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris N. Richardson, Johannes Ring, Marie E. Rognes, and Garth N. Wells. The FEniCS project version 1.5. *Archive of Numerical Software*, 3, 2015.

[3] Martin S. Alnaes, Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. Unified Form Language: A domain-specific language for weak formulations of partial differential equations. *ACM Transactions on Mathematical Software*, 40, 2014.

[4] D. N. Arnold and A. Logg. Periodic table of the finite elements, November 2014.

[5] Sylvio Bistafa. On the development of the Navier-Stokes equation by Navier. *Revista Brasileira de Ensino de Física*, 40, 11 2017.

[6] Mathew Causley and David Seal. On the convergence of spectral deferred correction methods. *Communications in Applied Mathematics and Computational Science*, 14(1):33–64, February 2019.

[7] Gerasimos Chourdakis, Kyle Davis, Benjamin Rodenberg, Miriam Schulte, Frédéric Simonis, Benjamin Uekermann, Georg Abrams, Hans-Joachim Bungartz, Lucia Cheung Yau, Ishaan Desai, Konrad Eder, Richard Hertrich, Florian Lindner, Alexander Rusch, Dmytro Sashko, David Schneider, Amin Totounferoush, Dominik Volland, Peter Vollmer, and Oguz Ziya Koseomur. preCICE v2: A sustainable and user-friendly coupling library. *Open Research Europe*, 2:51, April 2022.

[8] Gerasimos Chourdakis, David Schneider, and Benjamin Uekermann. OpenFOAM-preCICE: Coupling OpenFOAM with external solvers for multi-physics simulations. *OpenFOAM® Journal*, 3:1–25, Feb 2023.

[9] Alok Dutt, Leslie Greengard, and Vladimir Rokhlin. Spectral deferred correction methods for ordinary differential equations. *BIT Numerical Mathematics*, 40(2):241–266, Jun 2000.

[10] Matthew Emmett and Michael Minion. Toward an efficient parallel in time method for partial differential equations. *Communications in Applied Mathematics and Computational Science*, 7(1):105 – 132, 2012.

[11] S. Friedhoff, R. D. Falgout, T. V. Kolev, Scott P. MacLachlan, and Jacob B. Schroder. A Multigrid-in-Time Algorithm for Solving Evolution Equations in Parallel. In *Presented at: Sixteenth Copper Mountain Conference on Multigrid Methods, Copper Mountain, CO, United States, Mar 17 - Mar 22, 2013*, 2013.

[12] Martin J. Gander, Felix Kwok, and Bankim C. Mandal. Dirichlet–Neumann waveform relaxation methods for parabolic and hyperbolic problems in multiple subdomains. *BIT Numerical Mathematics*, 61(1):173–207, Mar 2021.

[13] G. W Griffiths and W. E. Schiesser. Linear and nonlinear waves. *Scholarpedia*, 4(7):4308, 2009. revision #154041.

[14] Robert C. Kirby. Algorithm 839: FIAT, a new paradigm for computing finite element basis functions. *ACM Transactions on Mathematical Software*, 30:502–516, 2004.

[15] Robert C. Kirby. FIAT: numerical construction of finite element basis functions. In Anders Logg, Kent-Andre Mardal, and Garth N. Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, volume 84 of *Lecture Notes in Computational Science and Engineering*, chapter 13. Springer, 2012.

[16] Robert C. Kirby and Anders Logg. A compiler for variational forms. *ACM Transactions on Mathematical Software*, 32, 2006.

[17] H. P. Langtangen and K.-A. Mardal. *Introduction to Numerical Methods for Variational Problems*. 2016.

[18] Hans Petter Langtangen and Anders Logg. *Solving PDEs in Python*. Springer, 2017.

[19] Jacques-Louis Lions, Yvon Maday, and Gabriel Turinici. Résolution d'edp par un schéma en temps «pararéel ». *Comptes Rendus de l'Académie des Sciences - Series I - Mathematics*, 332(7):661–668, 2001.

[20] Anders Logg, Kent-Andre Mardal, Garth N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.

[21] Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. FFC: the FEniCS form compiler. In Anders Logg, Kent-Andre Mardal, and Garth N. Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, volume 84 of *Lecture Notes in Computational Science and Engineering*, chapter 11. Springer, 2012.

[22] Anders Logg and Garth N. Wells. DOLFIN: automated finite element computing. *ACM Transactions on Mathematical Software*, 37, 2010.

[23] Anders Logg, Garth N. Wells, and Johan Hake. DOLFIN: a C++/Python finite element library. In Anders Logg, Kent-Andre Mardal, and Garth N. Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, volume 84 of *Lecture Notes in Computational Science and Engineering*, chapter 10. Springer, 2012.

[24] L.D. Marini and Alfio Quarteroni. An iterative procedure for domain decomposition methods: A finite element approach. 01 1986.

[25] Z E Musielak and B Quarles. The three-body problem. *Reports on Progress in Physics*, 77(6):065901, June 2014.

[26] Kristian B. Ølgaard and Garth N. Wells. Optimisations for quadrature representations of finite element tensors through automated code generation. *ACM Transactions on Mathematical Software*, 37, 2010.

[27] S.M.C.M Randiligama, Shiroshi Jayathilake, and Kushan Wijesundara. Applications of finite element method in structural engineering. 12 2016.

[28] Benjamin Rodenberg, Ishaan Desai, Richard Hertrich, Alexander Jaust, and Benjamin Uekermann. FEniCS–preCICE: Coupling FEniCS to other simulation software. *SoftwareX*, 16:100807, 2021.

[29] Benjamin Rüth, Benjamin Uekermann, Miriam Mehl, Philipp Birken, Azahar Monge, and Hans-Joachim Bungartz. Quasi-Newton waveform iteration for partitioned surface-coupled multiphysics applications. *International Journal for Numerical Methods in Engineering*, 122(19):5236–5257, 2021.

[30] Robert Speck. Algorithm 997: pySDC - prototyping spectral deferred corrections. *ACM Transactions on Mathematical Software (TOMS)*, 45, August 2019.

[31] Robert Speck, Daniel Ruprecht, Matthew Emmett, Michael Minion, Matthias Bolten, and Rolf Krause. A multi-level spectral deferred correction method. *BIT Numerical Mathematics*, 55(3):843–867, August 2014.

[32] Benjamin Uekermann, Hans-Joachim Bungartz, Lucia Cheung Yau, Gerasimos Chourdakis, and Alexander Rusch. Official preCICE adapters for standard open-source solvers. In *Proceedings of the 7th GACM Colloquium on Computational Mechanics for Young Scientists from Academia*, October 2017.

[33] Niklas Vinnitchenko. Evaluation of higher-order coupling schemes with FEniCS-preCICE. Master's thesis, Technical University of Munich, Jan 2024.

[34] O.C. Zienkiewicz, R. Taylor, and P. Nithiarasu. The finite element method for fluid dynamics: Seventh edition. *The Finite Element Method for Fluid Dynamics: Seventh Edition*, pages 1–544, 11 2013.