

Matrix-Free FEM Solver for the Advection-Diffusion-Reaction Equation

Samuele Allegranza, Vale Turco, Bianca Michielan, Valeriia Potrebiina

February 23, 2026

1 Introduction

This project implements a finite element solver for the advection-diffusion-reaction (ADR) equation in 2D and 3D using the deal.II library. The primary objective is to compare a matrix-free approach—leveraging sum factorization, SIMD vectorization, and geometric multigrid—against a traditional matrix-based approach using sparse matrix assembly. The comparison evaluates both computational performance (execution time) and memory usage across varying mesh refinement levels and Degrees of Freedom (DoFs).

2 Mathematical Background

2.1 Strong Formulation

The strong form of the Advection-Diffusion-Reaction equation is defined in the domain $\Omega \subset \mathbb{R}^d$ for $d \in \{1, 2, 3\}$:

$$-\nabla \cdot (\mu \nabla u) + \beta \cdot \nabla u + \gamma u = f \quad \text{in } \Omega \quad (1)$$

The boundary conditions are split into Dirichlet and Neumann boundaries:

- $u = g$ on $\Gamma_D \subset \partial\Omega$.
- $\nabla u \cdot \vec{n} = h$ on $\Gamma_N = \partial\Omega \setminus \Gamma_D$.

Where μ is the diffusion coefficient, β is the advection coefficient, γ is the reaction coefficient, and f is the forcing term.

2.2 Weak Formulation

The weak formulation consists of finding $u \in V$ such that:

$$a(u, v) = f(v) + a(Rg, v) \quad \forall v \in V \quad (2)$$

The bilinear form $a(u, v)$ and the linear functional $f(v)$ are defined as:

$$a(u, v) = \int_{\Omega} \mu \nabla u \cdot \nabla v + \int_{\Omega} (\beta \cdot \nabla u)v + \int_{\Omega} \gamma uv \quad (3)$$

$$f(v) = \int_{\Omega} fv + \int_{\Gamma_N} hv \quad (4)$$

2.3 Matrix-Free Discretization

Instead of assembling the global sparse matrix, the matrix-free method evaluates the action of the matrix on a vector on-the-fly. The local cell matrix operator is approximated using numerical quadrature:

$$(A_k)_{ij} \approx \sum_{q=1}^{N_q} [\mu(\nabla\varphi_j(x_q) \cdot \nabla\varphi_i(x_q)) + \beta \cdot \nabla\varphi_j(x_q)\varphi_i(x_q) + \gamma\varphi_j(x_q)\varphi_i(x_q)]|J_q|w_q \quad (5)$$

This allows the operator to be expressed in terms of tensor products and diagonal matrices:

$$A_k = B^T D_\mu B + B^T D_\beta K + K^T D_\gamma K \quad (6)$$

Where $B_{qj} = \nabla\phi_j(x_q)$ and $K_{qj} = \phi_j(x_q)$.

2.4 Non-Homogeneous Lifting

Because deal.II's Matrix-Free classes do not natively support non-homogeneous boundary problems, a lifting strategy was implemented. The system solves for $Au_0 = \hat{f}$, where $\hat{f} = f - Au_g$. The boundary conditions are handled by setting $u_g = g$ on the boundary nodes and 0 elsewhere.

3 Implementation Architecture

The solver relies on parameter files (`.prm` via deal.II's `ParameterHandler`) to dynamically configure problem parameters such as refinements, physical coefficients, forcing terms, and linear solver settings. The matrix-free implementation separates the action of the PDE operator from the overarching solver logic.

3.1 The Matrix-Free Operator (ADROperator)

The `ADROperator` class inherits from `MatrixFreeOperators::Base` and strictly encapsulates the action of the ADR matrix on a vector. It utilizes SIMD (Single Instruction, Multiple Data) vectorization for maximum computational throughput. Key methods include:

- `evaluate_coefficients()`: Precomputes the diffusion, advection, and reaction coefficients (μ, β, γ) at the quadrature points of each cell. These values are grouped into SIMD-ready `Table` structures (`mu_values`, `beta_values`, `gamma_values`) to prevent redundant mathematical evaluations during iterative solving.
- `local_apply()`: The core functional block implementing the cell-wise matrix multiplication. Utilizing deal.II's `FEEvaluation`, it reads the source vector's Degrees of Freedom (DoFs), evaluates values and gradients at quadrature points, applies the precomputed physical coefficients ($\mu\nabla u + \beta \cdot \nabla u + \gamma u$), integrates the result, and safely distributes it back to the global destination vector.
- `apply_add()`: Executes the global operator application by calling `MatrixFree::cell_loop()` over all cells, dispatching multi-threaded work to `local_apply()`.

- `compute_diagonal()` & `local_compute_diagonal()`: Computes and stores the inverse of the matrix diagonal elements. This extraction is essential for configuring the custom `JacobiSmoothen` used internally by the multigrid preconditioner.

3.2 The Matrix-Free Solver (ADRProblem)

The `ADRProblem` class orchestrates the execution of the Matrix-Free solver. Inheriting from `ADRParamHandler`, it processes user inputs and manages system lifecycle. Its primary responsibilities are broken down into:

- `setup_system()`: Initializes the finite element spaces, continuous mappings, and DoF handlers. Crucially, it constructs the geometric multigrid hierarchy by initializing a sequence of Matrix-Free operators (`mg_matrices`) mapped to the progressively coarser grid levels.
- `assemble_rhs()`: Computes the right-hand side of the linear system. This accounts for the analytical forcing terms, Neumann boundary fluxes, and handles the non-homogeneous Dirichlet lifting by subtracting Au_0 (where u_0 isolatedly contains the Dirichlet boundary values) from the source vector.
- `solve()`: Solves the algebraic system using Krylov subspace methods (either CG for symmetric diffusion-dominated problems or GMRES for advection-heavy non-symmetric problems). The solver is accelerated by a Geometric Multigrid preconditioner, which natively employs the `ADROperator`'s computed diagonal within a custom, matrix-free Jacobi smoother.

4 Benchmarks and Results

All benchmarks were executed on an Intel i7-14700HX @ 5.3GHz with 20 Cores and 28 Threads.

4.1 Measurement Methodology

To ensure a fair and accurate comparison between the matrix-based and matrix-free solvers, execution time and memory usage were rigorously profiled utilizing deal.II's built-in diagnostic libraries.

The computational time was tracked using deal.II's timing utilities, included via `<deal.II/base/timer.h>`. This allows for precise measurement of the setup, assembly, and linear solve phases across different mesh refinement levels and MPI ranks.

Memory statistics were gathered utilizing `<deal.II/base/memory_consumption.h>`. The measurement approach differs slightly depending on the solver architecture:

- For the **Matrix-Based** solver, memory usage inherently accounts for the heavy storage requirements of the global `SparseMatrix`, `SparsityPattern`, and the distinct sparse matrices and sparsity patterns required for each level of the geometric multigrid.
- For the **Matrix-Free** solver, memory consumption is explicitly calculated by overriding the `memory_consumption()` method within the `ADROperator` class. This

method calculates the memory occupied by the base Matrix-Free operator and adds the memory footprint of the cached SIMD coefficient tables (`mu_values`, `beta_values`, and `gamma_values`) that are evaluated at the quadrature points.

To facilitate automated benchmarking, both the `ADRMBProblem` and `ADRProblem` classes implement a `print_memory_usage()` method to log these statistics to the standard output, alongside a `print_memory_usage_to_file()` method which seamlessly exports the execution time and memory data into CSV files for external visualization.

4.2 2D Homogeneous Dirichlet Conditions

The 2D problem tested the exact solution $u_{ex} = \sin(2\pi x) \sin(2\pi y)$ on $\Omega = [0, 1]^2$. The matrix-free solver was tested on a single rank with varying advection coefficients ($\beta_{small} = [0.1 \ 0.3]^T$, $\beta_{medium} = [10 \ 30]^T$, $\beta_{big} = [20 \ 60]^T$).

As expected from theory, when $\|\beta\| \ll \mu$, the matrix A is almost symmetric, making the Jacobi smoother highly effective. Conversely, when $\|\beta\| \gg \mu$, A loses symmetry, reducing the Jacobi smoother's efficiency and drastically increasing computation time. As shown in table 1 convergence time degrades.

Refinement	DoFs	β_{small}	Time (s)	β_{medium}	Time (s)	β_{big}	Time (s)
5	4,225		0.54		0.62		0.94
6	16,641		2.04		2.43		3.46
7	66,049		8.41		9.64		14.40
8	263,169		35.74		44.12		57.82

Table 1: Matrix Free solver, single rank performance in 2D

4.3 3D Homogeneous Dirichlet Conditions

Using the exact solution $u_{ex} = \sin(2\pi x) \sin(2\pi y) \sin(2\pi z)$ and $\beta = [0.1 \ 0.2 \ 0.1]^T$, the matrix-based and matrix-free solvers were directly compared.

Unsurprisingly, memory consumption using the matrix-free solver is much lower compared to the matrix-based implementation. As can be seen in table 2, memory consumption is reduced by a factor of four almost consistently as DoFs increase. More surprisingly, computation time is much lower using matrix-free implementation, with a speedup of up to 15x at the highest refinement tested.

DoFs	MB Time (s)	MF Time (s)	MB Mem (MB)	MF Mem (MB)
4,913	17.14	0.83	4.94	1.28
35,937	133.98	6.76	38.29	9.92
274,625	1057.77	58.54	301.82	78.26
2,146,689	8448.95	553.39	2396.80	622.17

Table 2: Matrix Free Solver vs Matrix Based, single rank performance in 3D

To test scalability, the matrix-free solver was run across multiple MPI ranks on the refinements from 5 through 8. The results shown in table 3 can be compared with the results of the matrix based solver on single MPI (see table 2).

MPI Ranks	Time (s)	Memory (MB)
1	553.39	622.17
4	156.02	161.92
8	109.64	82.80
16	106.66	42.42
20	98.97	34.59

Table 3: Matrix Free solver, increasing MPI ranks

4.4 3D Non-Homogeneous Dirichlet Conditions

To validate the non-homogeneous lifting implementation, the solver was tested against the exact solution $u_{ex} = e^z \cos(2\pi x) \cos(2\pi y)$, with non-homogeneous Dirichlet conditions on the boundaries 2,3,4,5 and Neumann conditions on the boundaries 0,1 of a domain $\Omega = [0, 1]^3$, with the usual convention for boundary ids. Table 4 shows the results using the matrix-free solver.

DoFs	MB Time (s)	MF Time (s)	MB Mem (MB)	MF Mem (MB)
729	2.59	0.15	0.66	0.18
4,913	20.06	0.97	4.93	1.28
35,937	163.84	7.19	38.29	9.91
274,625	1075.50	64.74	301.81	78.26

Table 4: Matrix Free solver, single MPI ranks