POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Project Report

FEM Solver

Laurea Magistrale in High Performance Computing Engineering

Author: Federico Quartieri, Alessandro Ruzza, Daniele Salvi

Professor: Prof. Luca Formaggia

Academic year: 2025-2026

Contents

1 Introduction **2**

2 FEM theory [1] **3**

3 Workflow **4**
- 3.1 TOML Configuration Options . . . 4
  - 3.1.1 Problem Section . . . 5
  - 3.1.2 Equation Section . . . 5
  - 3.1.3 Boundary_Conditions (Array) . . . 5
  - 3.1.4 Time_Dependent Section . . . 5
  - 3.1.5 Quadrature Section . . . 5

4 Core files description **6**
- 4.1 `Point` . . . 6
- 4.2 `Cell` . . . 6
- 4.3 `Function` . . . 7
- 4.4 `Grid` . . . 7
- 4.5 `Quadrature` . . . 7
- 4.6 `Boundary Conditions` . . . 7
- 4.7 `FEM and FEM_td` . . . 7
- 4.8 Mains . . . 8
  - 4.8.1 TomlMain . . . 8
  - 4.8.2 Main . . . 8
  - 4.8.3 Main_td . . . 8

5 Parallelization **8**
- 5.1 OpenMP Strategy . . . 8
- 5.2 Key Parallelization Components . . . 8
  - 5.2.1 Thread-Safe Expression Evaluation . . . 8
  - 5.2.2 Parallel Matrix Assembly . . . 9
  - 5.2.3 Memory Management Strategy . . . 9

5.2.4 Thread Safety Verification . . . 10
5.3 Implementation Challenges and Solutions . . . 10
5.3.1 Challenge 1: Expression Parser Thread Safety . . . 10
5.3.2 Challenge 2: Sparse Matrix Assembly . . . 10
5.3.3 Challenge 3: Memory Efficiency . . . 10
5.4 Future Parallelization Extensions . . . 10

6 Compilation **10**
6.1 Static libraries . . . 10
6.2 Executables . . . 11
6.3 Tests . . . 11
6.4 Custom/utility targets . . . 11
6.5 Headers and third-party dependencies . . . 11

7 Software architecture **12**
7.1 Overview . . . 12
7.2 Core data structures . . . 12
7.3 Grid . . . 13
7.4 Quadrature . . . 14
7.5 Boundary Conditions . . . 14
7.6 FEM solvers . . . 15
7.7 Configuration and Main Driver . . . 16

8 Results evaluation **17**
8.1 Steady . . . 17
8.1.1 1D . . . 17
8.1.2 2D . . . 17
8.1.3 3D . . . 18
8.2 Time-Dependent . . . 19
8.2.1 1D . . . 19
8.2.2 2D . . . 19
8.2.3 3D . . . 19
8.3 Other results . . . 19

9 Performance evaluation **19**
9.1 Comments . . . 20

10 Conclusions **20**

1. Introduction

This project focuses on the development of a general-purpose Finite Element Method (FEM) solver within the High Performance Computing Engineering course. The solver provides a unified framework for solving elliptic (steady-state) and parabolic (time-dependent) partial differential equations in one, two, and three dimensions, making it applicable to a wide range of scientific and engineering problems.

We adopted a modular and extensible approach to the implementation of FEM. Each stage of the workflow (from input parsing to system solution) is clearly separated, so that new functionalities can be added without impacting the existing code base. The design emphasizes abstraction: grids, functions, boundary conditions, and quadrature rules are represented through generic class templates that can be instantiated in different dimensions. This structure makes it possible to handle 1D, 2D, 3D problems within a unified framework, while maintaining readability and reducing code duplication.

Flexibility is further enhanced by the use of a TOML configuration file, through which users can define forcing terms, diffusion, transport, and reaction coefficients, as well as boundary conditions, time and output options. This ensures reproducibility and allows complex test cases to be specified without modifying the source code.

To exploit modern computing resources, OpenMP parallelization has been integrated into the assembly and solution phases, improving performance on multi-core architectures. Finally, the solver outputs results in both CSV and VTU formats, enabling straightforward numerical analysis and visualization.

2. FEM theory [1]

The mathematical formulation considered is the standard reaction-diffusion equation.

In particular, given a generic $t \in (0,T]$, $T > 0$ and $\mathbf{x} \in \Omega \subset \mathbb{R}^{\dim}$, find $u = u(\mathbf{x},t) \in \mathbb{R}$ such that:

$$\begin{cases} \frac{\partial u}{\partial t} - \mu\Delta u + \mathbf{b}\cdot\nabla u + \sigma u = f, & \text{in } \Omega \times (0,T], \\ u(\mathbf{x},t) = g & \text{on } \Gamma_D \subseteq \partial\Omega \times (0,T], \\ \mu\nabla u \cdot \mathbf{n} = \phi, & \text{on } \Gamma_N \subseteq \partial\Omega \times (0,T], \\ u(\mathbf{x},t=0) = u_0, & \text{in } \Omega. \end{cases} \tag{1}$$

In the previous system:

- $u(\mathbf{x},t) \in \mathbb{R}$ represents the solution.
- $\mu = \mu(\mathbf{x}) \in \mathbb{R}$ is the diffusion term.
- $\mathbf{b} = \mathbf{b}(\mathbf{x}) \in \mathbb{R}^{\dim}$ is the transport term.
- $\sigma = \sigma(\mathbf{x}) \in \mathbb{R}$ is the reaction term.
- $f = f(\mathbf{x},t) \in \mathbb{R}$ is the forcing term.
- $g = g(\mathbf{x},t) \in \mathbb{R}$ is the dirichlet boundary condition, applied on the Dirichlet boundary Γ_D.
- $\phi = \phi(\mathbf{x},t) \in \mathbb{R}$ is the neumann boundary condition, applied on the Neumann boundary Γ_N.

In principle, there can be multiple Dirichlet and Neumann boundaries with different definitions of g and ϕ, although they cannot overlap.

The above is called the "strong formulation". To discretize, we must first obtain the "weak formulation", so that $\forall t \in (0,T]$ we find $u(\mathbf{x},t) \in V$

Let $V = H^1(\Omega)$ and consider $v \in V$. We multiply by v and integrate over Ω the first equation of the strong formulation:

$$\int_\Omega \frac{\partial c}{\partial t}\, v\, dx - \int_\Omega \mu\Delta u\, v\, dx + \int_\Omega \mathbf{b}\cdot\nabla u\, v\, dx + \int_\Omega \sigma u\, v\, dx = \int_\Omega f\, v\, dx. \tag{2}$$

Then, we integrate it by parts:

$$\int_\Omega \frac{\partial u}{\partial t}\, v\, dx + \int_\Omega \mu\nabla u\cdot\nabla v\, dx - \int_{\partial\Omega} \mu\nabla u\cdot\mathbf{n}\, v\, ds + \int_\Omega \mathbf{b}\cdot\nabla u\, v\, dx + \int_\Omega \sigma u\, v\, dx = \int_\Omega f\, v\, dx. \tag{3}$$

We can simplify the term related to the Neumann boundary condition:

$$\int_{\partial\Omega} \mu\nabla u\cdot\mathbf{n}\, v\, ds = \int_{\Gamma_N} \phi\, v\, ds. \tag{4}$$

So, with the weak formulation, $\forall t \in (0,T]$ we find $u(t) \in V$ such that:

$$\begin{cases} \int_\Omega \frac{\partial c}{\partial t}\, v\, dx + \int_\Omega \mu\nabla u\cdot\nabla v\, dx + \int_\Omega \mathbf{b}\cdot\nabla u\, v\, dx + \int_\Omega \sigma u\, v\, dx = \int_\Omega f\, v\, dx + \int_{\Gamma_N} \phi\, v\, dx & \forall v \in V, \\ u(\mathbf{x},t) = g(\mathbf{x},t) & \text{on } \Gamma_D \subseteq \partial\Omega \times (0,T], \\ u(\mathbf{x},t=0) = u_0(\mathbf{x}) & \text{on } \Omega. \end{cases} \tag{5}$$

We then discretize space-wise and time-wise. Let $\{\phi_i\}_{i=1}^N$ be the FE basis of V_h. Expand $u_h^n = \sum_{j=1}^N U_j^n\, \phi_j$ and test with $v_h = \phi_i$. The θ-scheme for linear diffusion−transport−reaction reads, for all $n = 0, 1, \ldots, N-1$ and for all i=1,...,N,

$$\frac{1}{\Delta t}\sum_{j=1}^N M_{ij}\left(U_j^{n+1} - U_j^n\right) \;+\; \theta\sum_{j=1}^N K_{ij}\, U_j^{n+1} \;+\; (1-\theta)\sum_{j=1}^N K_{ij}\, U_j^n \;=\; \theta\, F_i^{n+1} + (1-\theta)\, F_i^n,$$

with

$$M_{ij} = \sum_{K \in \mathcal{T}_h} \int_K \phi_j \, \phi_i \, dx,$$

$$K_{ij} = \sum_{K \in \mathcal{T}_h} \int_K \Big(\mu \, \nabla \phi_j \cdot \nabla \phi_i + (\mathbf{b} \cdot \nabla \phi_j) \, \phi_i + \sigma \, \phi_j \, \phi_i \Big) \, dx,$$

$$F_i^n = \sum_{K \in \mathcal{T}_h} \int_K f^{\,n} \, \phi_i \, dx \; + \; \sum_{e \subset \Gamma_N} \int_e \phi^{\,n} \, \phi_i \, ds.$$

Matrix form:

$$\Big(\tfrac{1}{\Delta t} M + \theta K \Big) U^{n+1} = \Big(\tfrac{1}{\Delta t} M - (1-\theta) K \Big) U^n + \theta F^{\,n+1} + (1-\theta) F^{\,n}.$$

with initial condition $u_h^0 = u_{0,h} \in V_h$

3. Workflow

The solver executes a structured sequence of steps that transforms the problem definition provided by the user into the numerical approximation of the PDE solution. The main stages of this workflow are:

1. **Configuration Parsing:** The simulation parameters are specified in a TOML configuration file. This file defines the problem dimension, the mathematical functions representing forcing, diffusion, transport, and reaction terms, the type and values of boundary conditions, specifications for time dependent problems and the desired quadrature order. The `Config` class is responsible for parsing the file and instantiating the corresponding objects.
2. **Mesh and Grid Setup:** Depending on the chosen dimension, the solver builds the computational grid using the dedicated `Grid` class. The grid stores nodes, elements, and boundary entities, and it supports both internally generated meshes (1D) and the parsing of external mesh files (e.g. `.msh` format, for 2D and 3D).
3. **Definition of Problem Functions:** Physical coefficients and source terms are represented as templated function objects, consistently across dimensions. Boundary conditions are organized through the `BoundaryConditions` class, which handle both Dirichlet and Neumann constraints.
4. **Quadrature Rules:** Numerical integration on elements and boundary faces is performed using dimension-dependent quadrature rules. The solver selects rules of sufficient order to guarantee accuracy for the chosen polynomial basis.
5. **Assembly of the Linear System:** For each element, local mass/stiffness matrices and load vectors are computed and then assembled into the global sparse system. The assembly phase incorporates variable coefficients and enforces boundary conditions.
6. **System Solution:** The resulting sparse linear system is solved using the Eigen library. The solution vector contains the nodal approximation of the unknown field.
7. **Output and Post-processing:** The solver exports the discrete solution for analysis and visualization in two formats:
   - **CSV**: tabular data suitable for quick plots and numerical post-processing (e.g., Python/NumPy, MATLAB). We export lines in format `x,y,z,solution`, only writing relevant coordinates (e.g. 1D output format is `x,solution`)
   - **VTU**: *VTK UnstructuredGrid* with the nodal scalar field `u`, intended for interactive visualization in ParaView. It can be used with standard filters (e.g., *Slice*, *Contour*, *Clip*, *Plot Over Line*) for inspection and figure generation.

`TomlMain.cpp` is the program entry point: it reads the user configuration (from a `.toml` file), understands whether the problem is steady or time-dependent and which dimension to use (1D,2D,3D), launches the corresponding solver, and saves the results in CSV and VTU. It does not do the numerical computations itself, it only directs the workflow.

3.1. TOML Configuration Options

Sections are declared by a `[section]` header (all lowercase).
Arrays are declared by a `[[array]]` header per element (all lowercase).

3.1.1 Problem Section

`dimension`	Problem dimensionality (1, 2, or 3)
`mesh_file`	Path to mesh file (.msh format)
`output_file`	Base name for output files
`1d_start`	Start coordinate for 1D uniform grids
`1d_end`	End coordinate for 1D uniform grids
`1d_size`	Number of grid points for 1D uniform grids
`time_dependent`	Enable time-dependent solving (true/false)

3.1.2 Equation Section

`diffusion_function`	Diffusion coefficient function $\mu(x, y, z)$
`transport_function_x`	Transport coefficient b in x-direction
`transport_function_y`	Transport coefficient b in y-direction
`transport_function_z`	Transport coefficient b in z-direction
`reaction_function`	Reaction coefficient function $\sigma(x, y, z)$
`forcing_function`	Right-hand side forcing term $f(x, y, z)$

3.1.3 Boundary_Conditions (Array)

`tag`	Boundary tag identifier (for 1D: start is 0, end is 1)
`type`	Condition type (`"dirichlet"` or `"neumann"`)
`function`	Boundary value function $g(x, y, z)$
`time_function`	Time-dependent boundary function $g(x, y, z, t)$

3.1.4 Time_Dependent Section

`final_time`	End time for time integration
`time_step`	Time step size Δt
`theta`	Theta scheme parameter (0=explicit, 0.5=Crank-Nicolson, 1=implicit)
`initial_condition`	Initial condition function $u_0(x, y, z)$
`forcing_function_td`	Time-dependent forcing function $f(x, y, z, t)$

3.1.5 Quadrature Section

`type`	Quadrature rule (`"order2"` or `"order4"`)

All function expressions support mathematical operations using variables `x`, `y`, `z`, and constants `pi`, `e`. Time-dependent functions also support variable `t`.

Listing 1: Example: Complete Time-Dependent Configuration

```
[problem]
dimension = 2
mesh_file = "../mesh/mesh-square-h0.050000_gmsh22.msh"
output_file = "output/complete_example_td"
time_dependent = true

[quadrature]
type = "order2"


[equation]
diffusion_function = "1.0"
transport_function_x = "0.0"
transport_function_y = "0.0"
transport_function_z = "0.0"
reaction_function = "x"
forcing_function = "0"
```

```
18
19 [time_dependent]
20 final_time = 1.0
21 time_step = 0.01
22 theta = 0.5
23 initial_condition = "0.0"
24 forcing_function_td = "x*y + x*x*y*t"
25
26 [[boundary_conditions]]
27 type = "neumann"
28 tag = 0
29 function = "0.0"
30 time_function = "-y*t"
31
32 [[boundary_conditions]]
33 type = "dirichlet"
34 tag = 1
35 function = "0.0"
36 time_function = "x*y*t"
37
38 [[boundary_conditions]]
39 type = "dirichlet"
40 tag = 2
41 function = "0.0"
42 time_function = "0.0"
43
44 [[boundary_conditions]]
45 type = "neumann"
46 tag = 3
47 function = "0.0"
48 time_function = "x*t"
```

4. Core files description

In order to better understand the internal structure of the solver, this section presents the most important files of the codebase. Each file plays a specific role in the implementation of the Finite Element Method, from the definition of basic data structures to the assembly of the global system and the management of boundary conditions. The following subsections provide an overview of the purpose and functionalities of these files.

4.1. Point

The file `point.hpp` defines the core data structure for representing spatial coordinates in the solver. The `Point` class is templated by dimension, making it reusable in 1D, 2D, and 3D. It stores coordinates in a lightweight array, overloads common operators for intuitive use, and provides utility functions such as norms, dot products, and distance computations. A notable feature is the templated `operator*`, which unifies scalar-vector and vector-scalar multiplications as well as the dot product of vectors with equal dimension.

4.2. Cell

The file `cell.hpp` defines the data structure used to represent mesh elements, or *cells*, in the finite element solver. A cell stores the global indices of the nodes at its vertices and the corresponding `Point` objects that define its geometry. Exactly as for the points, this class is also templated by dimension so that it can represent line segments in 1D, triangles in 2D, and tetrahedra in 3D.
Besides holding connectivity information, the class provides utility methods for computing geometric quantities such as edge lengths, areas, or volumes, as well as mapping barycentric coordinates for interpolation.
Boundary entities are represented by a derived BoundaryCell data structure, a d-dimensional element embedded in d+1 dimensions (point in 1D, edges in 2D, triangular faces in 3D). It inherits the geometry from `Cell<d+1>` and adds a boundary tag `boundary_id` to label parts of $\partial\Omega$. It overrides functions that provide mappings and geometric quantities on these elements to adapt them to the lower dimension of the boundary element.

4.3. Function

The files `function.hpp` and `function.tpp` define the `Function` template, used to represent fields $f : \mathbb{R}^{\text{dim}} \to \mathbb{R}^{\text{returnDim}}$ over the computational domain.
The template is parameterized by spatial input and output dimensions, allowing the same interface to handle both scalar and vector-valued data. Evaluation methods return the function value at a given `Point`, providing a uniform way to access all problem coefficients during assembly and boundary condition enforcement. It is used uniformly for source terms, diffusion and reaction coefficients, transport velocities, and boundary/initial data. Functions can be constant or spatially varying and are typically constructed from the configuration.
This abstraction decouples assembly and boundary-condition code from how coefficients are defined, making them easy to swap or parameterize.

Time-dependent functions are also defined in `function.hpp` as `fun_td`, also templated on spatial input and output dimensions. Templated helper functions manage casting between `Function` and `fun_td`

4.4. Grid

The files grid.hpp and grid.cpp implement the mesh management system that serves as the spatial discretization foundation for the finite element solver. The Grid class is templated by dimension and provides a unified interface for handling computational meshes across 1D, 2D, and 3D domains. It stores and manages collections of volume elements (cells), boundary elements (boundary cells), and nodal coordinates, providing efficient access methods for mesh traversal during assembly.
The Grid supports reading mesh data from GMSH format files (.msh version 2.2) through the `parseFromMsh` method, which parses node coordinates and element connectivity information while distinguishing between volume and boundary elements based on their element types. Physical boundary tags from the mesh file are preserved and used for boundary condition assignment. The class provides comprehensive query methods for accessing boundary nodes and cells by physical tags, enabling integration with the boundary condition system. For 1D problems, uniform grids can be generated programmatically, while 2D and 3D geometries rely on external mesh files.

4.5. Quadrature

The files `quadrature.hpp` and `quadrature.cpp` implement the quadrature rules [2] used to approximate all volume and surface integrals required by the solver. They provide rules with selectable order for the reference elements in each dimension (segments in 1D, triangles in 2D, tetrahedra in 3D) and for their faces (points in 1D, segments in 2D, triangles in 3D).

The quadrature point values and weights were obtained using the scikit-fem python library. [3]
In practice, assembly loops traverse quadrature points to evaluate basis functions/gradients, sample problem coefficients, apply the Jacobian, and accumulate contributions to element matrices and boundary terms. The quadrature order tunes accuracy versus cost, and the module leverages templates (by dimension/element) to expose a uniform, compile-time efficient interface across 1D−3D. Face quadrature points are also used to evaluate boundary dataand outward normals on boundary entities, enabling consistent BC treatment across dimensions.

4.6. Boundary Conditions

These files implement the handling of boundary conditions for both steady and time-dependent problems. They provide a dimension-templated interface to register conditions by mesh boundary tags and to apply them consistently during assembly/solution. Dirichlet constraints are enforced strongly by modifying the global system (LHS matrix rows/columns and RHS) at the involved DOFs, while Neumann data are incorporated as boundary integrals on faces using the face-quadrature points exposed by the quadrature and grid modules. Note that Neumann contributions are applied first, then Dirichlet constraints are enforced, to ensure Dirichlet conditions take precedence over Neumann.

The steady module evaluates user-defined functions for Dirichlet values and for Neumann fluxes and applies them once. The time-dependent counterpart extends the same API re-evaluating and reapplying BCs at each time step. They are not applied on the initial condition, as that is interpreted as exact on every point of the domain (boundaries included).

4.7. FEM and FEM_td

The files `fem.hpp, fem.cpp` and `fem_td.hpp, fem_td.cpp` provide the core solvers for steady (elliptic) and time-dependent (parabolic) problems, templated by spatial dimension so the same interfaces serve 1D, 2D and

3D. Each solver owns the mesh and problem fields (forcing, diffusion, transport, reaction), interfaces with the boundary-condition layer, solves sparse systems with Eigen,(`SparseLU` (direct) or `BiCGSTAB` (iterative), adapting on the number of non-zero elements of the final system), exploits OpenMP over element loops, and writes CSV/VTU outputs.
In the time-dependent case, time-invariant operators are assembled once ($A = M/\Delta t + \theta * K$). Then, at each step, time-varying coefficients (forcing term) and BCs are evaluated at t_{n+1} and applied. The system is solved, the solution for that time step is written to `.csv` and `.vtu`, and the state advanced. Initial conditions are set before the first time step.

4.8. Mains

The solver provides three alternative entry points, each for a different workflow and user experience.

4.8.1 TomlMain

`TomlMain.cpp` is the configuration-driven entry point of the solver. It reads a user-provided `.toml` file, validates its sections, and uses the `Config` module (`config.hpp`, `config.cpp`) to instantiate all problem objects. The `Config` layer parses TOML keys and maps them to concrete objects for the FEM pipeline: it builds the grid (from a Gmsh `.msh` file for 2D,3D or parameters for 1D), wraps the problem coefficients as `Function<dim,returnDim>` or `fun_td<dim,returnDim>` for time-dependent terms, and assembles the boundary condition registry by associating mesh tags with Dirichlet or Neumann conditions.
The configuration file was explained in section 3.1. This TOML-driven setup separates experiment specification from code, making runs reproducible and easily adjustable without recompilation.

4.8.2 Main

`main.cpp` provides a hard-coded entry point for steady-state problems. Instead of reading a configuration file, key problem parameters—such as dimension and mesh file—are set directly in the code or passed via command line. This mode is useful for quick tests and controlled debugging, bypassing TOML parsing and focusing on core FEM routines.

4.8.3 Main_td

`main_td.cpp` is a hard-coded entry point for time-dependent problems. Similar to `main.cpp`, it sets parameters and initial conditions programmatically or via command line, allowing direct experimentation with transient solvers without the need for a full TOML configuration. Like `main.cpp`, this approach is suited for development, debugging, or educational use.

5. Parallelization

5.1. OpenMP Strategy

The project implements **shared-memory parallelization** using OpenMP to accelerate the computationally intensive finite element assembly process. The parallelization strategy focuses on **element-wise parallel loops** with careful attention to thread safety and memory management.

5.2. Key Parallelization Components

5.2.1 Thread-Safe Expression Evaluation

The most critical parallelization challenge was handling mathematical expression parsing in multi-threaded environments. The solution employs a **ThreadExpressionPool** pattern:

Listing 2: Thread-safe expression evaluation

```
class ThreadExpressionPool {
    std::vector<double> x_vals, y_vals, z_vals, t_vals;
    std::vector<exprtk::expression<double>> expressions;
    std::vector<exprtk::symbol_table<double>> symbol_tables;
public:
    ThreadExpressionPool(int numThreads) {
        // [...] Also resize x,y,z,t_vals vectors
        expressions.resize(numThreads); // One expression per thread
```

```
        symbol_tables.resize(numThreads);
    }
};
// In code, construct thread-safe function using thread expression pool
pool = std::make_shared<ThreadExpressionPool>(num_threads);
[pool](Point<dim> p) -> double {
    #ifdef _OPENMP
        const int thread_id = omp_get_thread_num();
    #else
        const int thread_id = 0;
    #endif
    // Update variables for this thread
    pool->x_vals[thread_id] = (dim >= 1) ? p[0] : 0.0;
    pool->y_vals[thread_id] = (dim >= 2) ? p[1] : 0.0;
    pool->z_vals[thread_id] = (dim >= 3) ? p[2] : 0.0;

    return pool->expressions[thread_id].value();
};
```

This eliminates race conditions by providing each thread with its own `exprtk` expression evaluator, accessed via `omp_get_thread_num()`.

5.2.2 Parallel Matrix Assembly

The finite element assembly process parallelizes the loop over mesh elements:

Listing 3: Time-dependent parallel assembly

```
#ifdef _OPENMP
#pragma omp parallel
{
    // separate preallocated vectors: thread safety + memory efficiency
    int tid = omp_get_thread_num();
    std::vector<Triplet>& tM = tripletM_thr[tid];
    std::vector<Triplet>& tK = tripletK_thr[tid];
    VectorXd& F_local = F_threads[tid];

    #pragma omp for schedule(dynamic, 32)           // Dynamic load distribution
    for (int e = 0; e < numElements; ++e) {
        assemble_M_and_K_element(e, tM, tK);        // Mass and stiffness
        assemble_forcing_element(e, F_local, t);    // Time-dependent RHS
    }
}

// Merge thread-local contributions
for (auto& v : tripletM_thr) tripletM.insert(tripletM.end(), v.begin(), v.end());
for (auto& v : tripletK_thr) tripletK.insert(tripletK.end(), v.begin(), v.end());
for (int tid = 0; tid < nthreads; ++tid) f_new += F_threads[tid];
#endif
```

Key design decisions:

- **Dynamic scheduling**: Handles load imbalancing from varying element sizes
- **Thread-local storage**: Each thread maintains private temporary matrices
- **Serialized global assembly**: Protect global matrix assembly operations

5.2.3 Memory Management Strategy

To minimize memory allocation overhead and prevent race conditions:

- **Pre-allocated thread-local vectors**: Each thread has dedicated storage for local contributions
- **Reserve capacity**: Sparse matrix triplets pre-reserve memory based on mesh connectivity
- **Reuse storage**: Thread-local matrices are cleared and reused across elements

Listing 4: Memory management for parallel assembly

```
// In class constructor
F_threads.resize(omp_get_max_threads());
for (auto& F_local : F_threads) {
    F_local.resize(grid.getNNodes());
}

// In parallel assembly
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    VectorXd& F_local = F_threads[tid];
    F_local.setZero();  // Reuse pre-allocated memory
}
```

5.2.4 Thread Safety Verification

- **Race condition elimination**: No shared data modifications in parallel regions
- **Deterministic results**: Parallel execution produces identical results to serial
- **Memory safety**: Thread-local storage prevents memory corruption

5.3. Implementation Challenges and Solutions

5.3.1 Challenge 1: Expression Parser Thread Safety

Problem: `exprtk` library is not thread-safe when multiple threads access the same expression object.
Solution: Thread-local expression pools with one parser per thread, accessed via thread ID.

5.3.2 Challenge 2: Sparse Matrix Assembly

Problem: Concurrent writes to global sparse matrices cause race conditions.
Solution: Thread-local triplet accumulation followed by efficient matrix construction via the `setFromTriplets()` library method.

5.3.3 Challenge 3: Memory Efficiency

Problem: Thread-local storage increases memory footprint significantly.
Solution: Careful memory reuse and pre-allocation strategies to minimize overhead.

5.4. Future Parallelization Extensions

- **Distributed memory**: MPI integration for multi-node scaling
- **GPU acceleration**: CUDA/OpenCL support for assembly kernels
- **Hybrid parallelization**: MPI + OpenMP for large-scale problems
- **Task-based parallelism**: OpenMP tasks for irregular workloads

The OpenMP implementation successfully achieves **good parallel efficiency** while maintaining **numerical accuracy** and **thread safety**, making it suitable for computations on shared-memory systems.

6. Compilation

From the repository root, to compile the project create and enter a build directory with `mkdir build && cd build`, then configure with `cmake ..` and compile with `make` (optionally, `make -j N` for parallel compilation on N threads). The libraries and executables will be generated in `build/`. The program output is written by default to `build/output/`.

6.1. Static libraries

Many template implementations are placed in `.cpp` files (rather than `.tpp` headers) to enable true separate compilation and reuse through the static library. With header-only (`.tpp`) templates, every translation unit (all executables and tests) re-compiles the same template code, inflating build times and object sizes. By moving definitions into `.cpp` and compiling them once into `fem1d_lib`, all consumers simply link the already-compiled

code. The trade-off is that we must explicitly instantiate the templates we need; in our project we only use `dim` $\in \{1, 2, 3\}$, so writing those explicit instantiations is a small, predictable cost for faster and more consistent builds across multiple targets.

`fem1d_lib` (always built)

Sources: all `src/*.cpp` except `config.cpp`, `TomlMain.cpp`, `main.cpp`, `main_td.cpp` (i.e., the core FEM implementation without entry points).
Linked libraries: `Eigen3::Eigen` (required) and `OpenMP::OpenMP_CXX` (if found).

`fem1d_lib_sequential` (built only if OpenMP is found)

Same sources as `fem1d_lib`.
Linked libraries: `Eigen3::Eigen` (no OpenMP linkage).
Purpose: enable side-by-side comparison of parallel vs. sequential builds.

6.2. Executables

The executables depend on custom target `create_output_dir` to ensure `build/output/` exists before the build).

`17_fem1d_17_fem1d` (always built)

Source: `src/main.cpp`. Linked to `fem1d_lib`.

`TomlMain` (always built)

Sources: `src/TomlMain.cpp` and `src/config.cpp`. Linked to `fem1d_lib`.
Intended for TOML-driven runs.

`sequentialTomlMain` (built only if OpenMP is found)

Sources: `src/TomlMain.cpp` and `src/config.cpp`. Linked to `fem1d_lib_sequential`.
Purpose: sequential reference for speedup analysis.

`17_fem1d_17_fem1d_td` (always built)

Source: `src/main_td.cpp`. Linked to `fem1d_lib`.

6.3. Tests

`17_fem1d_17_fem1d_test` (always built)

Sources: all `test/*.cpp`. Linked to `fem1d_lib` and `gtest_main`.
Tests are registered with CTest via `gtest_discover_tests`, enabling automatic discovery and execution with `ctest`.

6.4. Custom/utility targets

`create_output_dir` (always built)

Creates the `output/` directory in the build tree.

`empty_output_dir` (always built)

Removes any `output/*.csv` and `output/*.vtu` files from the build directory.

6.5. Headers and third-party dependencies

Header search paths include `include` and `include/external` (e.g., `toml.hpp`, `exprtk.hpp`).
Eigen3 (≥ 3.3) is required and linked via `Eigen3::Eigen`.
OpenMP is optional. If found, parallel support is enabled in `fem1d_lib` and an additional sequential pair (`fem1d_lib_sequential`, `sequentialTomlMain`) is produced for performance comparisons.

7. Software architecture

7.1. Overview

The codebase is organized in modular components that mirror the FEM workflow: configuration → mesh/geometry → fields and quadrature → assembly → boundary conditions → linear solve → output. Templates by spatial dimension ensure a single code path serves 1D/2D/3D with minimal duplication.

7.2. Core data structures

`Point<dim>` (point.hpp).

- **Role** — Represents a point in N-dimensional space and provides basic geometric and algebraic operations for FEM and related computations.
- **Main Features**
  - **Data Storage**
    - Uses a `std::vector<double>` to store coordinates for flexibility.
  - **Static Methods**
    - `zero()` — returns a point at the origin.
    - `one()` — returns a point with all coordinates set to 1.
  - **Constructors**
    - From a vector or array of coordinates.
    - Default constructor (origin).
    - From 1, 2, or 3 doubles (for 1D, 2D, 3D points only).
  - **Type Conversion**
    - Implicit/explicit cast to `double` for `Point<1>`.
  - **Access Methods**
    - `operator[]` — access coordinate by index (with bounds check).
    - `x()`, `y()`, `z()` — access first, second, third coordinate (if dimension allows).
  - **Operators**
    - Addition and subtraction between points.
    - Scalar multiplication: `Point * double`.
    - Distance calculation between points.
  - **Templated `operator*`**
    - Handles scalar-vector and vector-scalar multiplication when one operand is 1D.
    - Computes the dot product when both operands have the same dimension.
    - Returns `Point<1>` for the dot product and a `Point<dim>` for scalar-vector cases.
    - Emits a compile-time error for incompatible dimensions.
  - **Usage**
    - Used across the codebase to represent mesh nodes, quadrature points, gradients, and other geometric entities, providing a unified interface in any dimension.

`Cell` Module (cell.hpp / cell.cpp).

- **Role** — Represents mesh elements (cells) and boundary elements (faces/edges) in arbitrary dimension, providing geometry, connectivity, and shape function utilities for FEM assembly.
- `Cell<dim>`
  - Stores node coordinates (`nodes`) and global indices (`nodeIndices`).
  - Provides access to node coordinates and indices.
  - **Methods:**
    - `getN()` – number of nodes.
    - `operator[]`, `getNode(i)` — access node by local index.
    - `getNodeIndex(i)`, `getNodeIndexes()` — access global node index(es).
    - `mapToGlobal(barycentricCoords)` — map barycentric coordinates to global coordinates.
    - `barycentricGradient(i)` — gradient of the i-th barycentric shape function (dimension-specialized).
    - `measure()` — element measure (length/area/volume, dimension-specialized).
- `BoundaryCell<dim>`
  - Inherits from `Cell<dim+1>`.
  - Adds a boundary tag `boundary_id` to identify the physical boundary.
  - **Methods:**
    - `getBoundaryId()` — return the boundary tag.

  - `mapToGlobal(barycentricCoords)` — mapping for the boundary element.
  - `measure()` — measure of the boundary element (length/area, dimension-specialized).
- **Implementation Details (cell.cpp)**
  - **Specializations for `measure()`:**
    - **1D** — length of a segment.
    - **2D** — area of a triangle.
    - **3D** — volume of a tetrahedron.
  - **Specializations for `barycentricGradient(i)`:**
    - **1D** — ± 1/length for the two nodes.
    - **2D** — constant gradient vector for each triangle node.
    - **3D** — constant gradient vector for each tetrahedron node (via face normals).
  - **Specializations for `BoundaryCell<dim>::measure()`:**
    - **1D** — length of an edge in 2D.
    - **2D** — area of a face in 3D.
- **Usage** — Used throughout the FEM codebase to represent elements and boundary faces/edges, providing the geometric and topological information required for local matrix assembly and quadrature.

`Function` **Module (function.hpp / function.tpp).**
- **Role** — Provides a flexible, templated class to represent mathematical functions (scalar or vector-valued) over a domain of arbitrary dimension, supporting algebraic operations, derivatives, and time-dependent extensions.
- **Main Types**
  - `Function<dim, returnDim>`
    - Represents a function from `Point<dim>` to `Point<returnDim>`.
    - Internally stores a `std::function<Point<returnDim>(const Point<dim>&)>`.
    - Static utilities: `zeroFun`, `oneFun` (constant zero/one functions).
    - **Constructors:** from a callable/lambda.
    - **Operators:** `operator+`, `operator*` (with functions or scalars), `operator+=`.
    - **Evaluation:** `operator()`, `value`.
    - **Time interface:** conversion to `fun_td<dim, returnDim>`; utility `castToSteadyFunction`.
  - `Function<dim, 1>` (specialization)
    - Adds support for derivatives (gradient) stored as a vector of functions.
    - **Constructors:** function plus up to three derivative functions (by dimension).
    - **Derivative evaluation:** `dx_value`, `dy_value`, `dz_value`, `getGradValues`.
    - **Gradient as function:** `getGrad` (returns a vector-valued function).
- **Derived Classes**
  - `ZeroFunction`, `OneFunction` — constant zero/one functions for any dimension.
- **Implementation Details (function.tpp)**
  - Implements operators (addition, multiplication, scalar operations) for general and specialized cases.
  - Propagates derivatives correctly for `Function<dim, 1>`.
  - Ensures efficient and thread-safe evaluation.
- **Usage**
  - Represents coefficients, source terms, and boundary data in the FEM codebase.
  - Supports analytical expressions and user-defined lambdas.
  - Extensible to time-dependent problems via `fun_td` and related utilities.

7.3. Grid

`Grid` **Module (grid.hpp / grid.cpp).**
- **Role** — Provides mesh management and spatial discretization for finite element computations in arbitrary dimensions, handling both volume elements and boundary entities with support for external mesh file formats.
- `Grid<dim>`
  - Template class for mesh representation in arbitrary dimension `dim`.
  - **Stores:**
    - Volume elements (`CellVector cells`).
    - Boundary elements (`BoundaryCellVector boundary_cells`).
    - Node coordinates (`NodeVector unique_nodes`).
  - **Methods:**
    - Constructors — from pre-built cell and node vectors, with optional boundary cells.

  - Getters — for elements, nodes and boundary cells
  - `getBoundaryNodesByTag(id)`, `getBoundaryCellsByTag(id)` — queries for BC application.
  - `parseFromMsh(filename)` — reads Gmsh mesh files (`.msh` v2.2) with physical boundary tags.
- **Implementation Details (grid.cpp)**
  - **File Parsing** — `parseFromMsh` processes Gmsh format, distinguishing volume and boundary elements by element type.
  - **Element Classification** — `isBoundaryCell(elemType)` identifies boundary vs. volume elements (dimension-specialized).
  - **Connectivity Parsing** — `parseBoundaryCell` and `parseInternalCell` extract node indices and physical tags.
  - **Tag Management** — Preserves physical boundary tags for consistent boundary condition assignment.
- **Usage**
  - For 2D/3D problems: load meshes from Gmsh files via `parseFromMsh()`.
  - For 1D problems: construct uniform grids programmatically using `Grid1D`.
  - Access mesh entities during assembly loops and boundary condition application.
  - Query boundary nodes/cells by physical tags for seamless integration of BCs.

7.4. Quadrature

quadrature.hpp / quadrature.cpp

- **Role** — Provides classes and methods for numerical integration (quadrature rules) over finite elements in 1D, 2D, and 3D, including both element interiors and boundaries. Used for assembling local matrices and vectors in finite element methods.
- **Main Classes**
  - `QuadratureRule<dim>`
    - Abstract base class for quadrature rules on simplices (segments, triangles, tetrahedra).
    - Stores barycentric coordinates of quadrature points and their weights.
    - Provides `getQuadratureData` to compute quadrature points, weights, shape functions, and gradients for a given cell.
    - Not intended to be instantiated directly.
  - `OrderTwoQuadrature<dim>` and `OrderFourQuadrature<dim>`
    - Derived from `QuadratureRule<dim>`.
    - Implement quadrature rules of order 2 and 4 for different dimensions.
    - Constructors set up the correct barycentric points and weights. [3]
  - `GaussLegendre<dim>`
    - Quadrature rules for boundary segments (edges/faces for 2D/3D).
    - Stores quadrature points and weights.
    - Provides `getQuadratureData`, and `integrateShapeFunctions`.
- **Key Methods**
  - `getQuadratureData` — Computes quadrature points, shape function values, and weights for a given cell or boundary.
  - `integrateShapeFunctions` (`GaussLegendre`) — Integrates shape functions times a given function (e.g., Neumann data) on a boundary, stores resulting contributions in a vector passed by reference.
- **Implementation Details**
  - Quadrature points defined in barycentric coordinates and mapped to global coordinates via cell `mapToGlobal()` method.
  - Weights scaled by measure (length, area, volume) of the element or boundary.
  - Specialized constructors per dimension and order (Gauss-Legendre in 1D, symmetric rules for 2D/3D).
- **Usage**
  - Instantiate the appropriate quadrature class for element or boundary type and order.
  - Call `getQuadratureData` to obtain integration data.
  - Use `integrateShapeFunctions` for direct numerical integration tasks.

7.5. Boundary Conditions

boundary_conditions.hpp / boundary_conditions.cpp

- **Role** — Provides classes and methods to define, store, and apply Dirichlet and Neumann boundary conditions for finite element problems in 1D, 2D, and 3D.
- **Main Classes**

  - `BoundaryCondition<dim, returnDim>`
    - Represents a single boundary condition.
    - Stores:
      - Boundary/tag ID (integer).
      - Condition type (`BCType`: Dirichlet or Neumann).
      - Function describing the boundary value (constant or general function).
    - **Constructors:**
      - From a function.
      - From a constant value (convenience).
  - `BoundaryConditions<dim, returnDim>`
    - Manages a collection of `BoundaryCondition` objects.
    - Provides methods to add Dirichlet or Neumann conditions (functions or constants).
    - Main method: `apply`, which enforces all stored conditions on the system matrix and right-hand side vector.
- **Key Methods**
  - `addDirichlet`, `addNeumann` — add boundary conditions for a given tag, from function or constant.
  - `apply` — applies all boundary conditions. Dirichlet conditions are applied after Neumann conditions for efficiency.
  - `applyDirichlet` — for each node on the boundary: sets matrix row to zero, diagonal to one, and RHS to boundary value. Parallelized with OpenMP.
  - `applyNeumann` — integrates Neumann data on boundary faces/edges using quadrature and adds contributions to the RHS. Specialized for 1D.
- **Implementation Details**
  - Uses the `Function` class to represent boundary data (both constants and general functions).
  - Employs quadrature rules (e.g., Gauss-Legendre) for Neumann integration.
  - Unified treatment of Dirichlet and Neumann conditions, with clear separation of logic.
  - Parallelization via OpenMP and thread-local storage for higher dimensions.
- **Usage**
  - Create a `BoundaryConditions` object.
  - Add Dirichlet and/or Neumann conditions using the provided methods.
  - Call `apply` to enforce conditions on the system before solving.

boundary_conditions_td.hpp / boundary_conditions_td.cpp
- **Role** — Provides the time-dependent counterpart of the Boundary Conditions module, supporting functions of both space and time.
- **Function Type**
  - Uses `fun_td<dim, returnDim>` (functions of space and time).
  - Constructors and `addDirichlet` / `addNeumann` methods accept time-dependent functions or constants.
- **BoundaryCondition_td<dim, returnDim>**
  - Stores a time-dependent function.
  - Provides `getBoundaryFunction(double t)` to obtain a steady (spatial-only) function at time `t`.
- **Application Methods**
  - `apply`, `applyDirichlet`, `applyNeumann` all require an extra argument `double t`.
  - At application time, the boundary data is evaluated at the given time by freezing the function.
- **Usage**
  - When applying boundary conditions, the current time `t` must be provided.
  - Logic, structure, and parallelization are otherwise nearly identical to the steady version.

7.6. FEM solvers

FEM Module (fem.hpp / fem.cpp).
- **Role** — Implements the core finite element method (FEM) solver for scalar problems in 1D, 2D, and 3D, including matrix assembly, boundary condition application, and solution output.
- **Main Class**
  - `Fem<dim>`
    - Template class for FEM in arbitrary dimension `dim`.
    - **Stores:**
      - Computational mesh (`Grid<dim>`).
      - Problem coefficients: forcing term, diffusion, transport, reaction (all as `Function` objects).

  - Boundary conditions (`BoundaryConditions<dim, 1>`).
  - `QuadratureRule<dim>` for integration.
  - System matrix (`SparseMat`), right-hand side (`VectorXd`), and solution vector (`VectorXd`).
- **Key Methods**
  - `Constructor` — initializes problem data, allocates system matrix and vectors. Accepts coefficients, boundary conditions, and quadrature as arguments.
  - `assemble` — assembles the global system matrix and RHS; loops over all elements, computes local contributions with quadrature, and inserts into global system. Applies boundary conditions after assembly. Supports OpenMP parallelism.
  - `assembleElement` — computes local element matrices/vectors for diffusion, transport, reaction, and forcing terms. Uses quadrature points and weights.
  - `solve` — solves the linear system using either direct solver (`SparseLU`) or iterative solver (`BiCGSTAB`), depending on size. Reports solver status.
  - `outputVtu` — writes solution and mesh to a VTU file (for ParaView visualization).
  - `outputCsv` — writes solution and mesh to a CSV file (for analysis).
- **Implementation Details**
  - Uses Eigen for sparse matrices and linear algebra.
  - Designed for extensibility to higher dimensions and complex PDEs.
  - Parallelized assembly and boundary application with OpenMP.
- **Usage**
  - Instantiate `Fem<dim>` with mesh, coefficients, boundary conditions, and quadrature.
  - Call `assemble()` to build the system.
  - Call `solve()` to compute the solution.
  - Export results with `outputVtu()` or `outputCsv()`.

FEM_td key differences (fem_td.hpp / fem_td.cpp).
- **Role** — Implements the time-dependent (parabolic) FEM solver, supporting evolution in time with the θ-method, in contrast to the steady-state solver in `fem`.
- **System Matrices**
  - Assembles and stores both mass matrix M and stiffness matrix K (in addition to A), required for time integration.
  - `fem` only assembles the stiffness matrix A for steady problems.
- **Time Integration**
  - Implements a time-stepping loop (`run`) and a single-step solver (`step`) using the θ-method.
  - `fem` only solves a single linear system without time-stepping.
- **Initial Condition**
  - Supports setting and applying an initial condition $u_0(x)$.
  - `fem` does not require or use an initial condition.
- **Forcing Term**
  - Accepts time-dependent forcing $f(x, t)$, with assembly at each time step.
  - `fem` uses only a spatial forcing $f(x)$.
- **Boundary Conditions**
  - Uses `BoundaryConditions_td`, which require the current time t when applying boundary conditions.
  - `fem` uses time-independent boundary conditions.
- **Output**
  - Can output the solution at each time step (CSV/VTU), enabling time-dependent visualization.
  - `fem` outputs only the steady-state solution.
- **Usage**
  - Designed for parabolic/time-dependent PDEs (e.g. heat equation).
  - `fem` is for elliptic/steady-state problems (e.g. Poisson).

7.7. Configuration and Main Driver

config.hpp / config.cpp / TomlMain.cpp
- **Role** — Handles reading and parsing the TOML configuration file, setting up all problem data (mesh, coefficients, boundary conditions, time settings, etc.), and launching the appropriate FEM solver (steady or time-dependent).
- **Key Components**
  - `Config Struct (config.hpp / config.cpp)`

  - Central structure holding all problem, equation, boundary, and time-dependent settings.
  - Sub-structures for: problem setup, coefficients (expressions), boundary conditions, time parameters, and quadrature.
  - Provides `loadFromFile` to parse a TOML file and populate fields.
  - Includes validation and pretty-printing utilities.
- `Factory Methods (config.hpp / config.cpp)`
  - Create objects required for the FEM solver:
    - Mesh/grid (1D, 2D, 3D).
    - Forcing, diffusion, reaction, and transport functions (via exprtk expressions).
    - Boundary conditions (steady and time-dependent).
    - Quadrature rules (order 2 or 4).
    - Initial conditions and unsteady forcing terms.
  - Use thread-safe expression pools (exprtk) for OpenMP parallelism.
- `Main Driver (TomlMain.cpp)`
  - Reads config file from command line.
  - Detects whether problem is steady or time-dependent.
  - Calls solver function for the chosen dimension (1D, 2D, 3D).
  - Each solver:
    - Instantiates all problem data using factory methods.
    - Constructs FEM (steady or time-dependent) object.
    - Runs assembly, solution, and output.
    - For unsteady problems: sets up time loop, initial condition, and time-dependent forcing.

- **Implementation Details**
  - Supports both constant and function-based coefficients.
  - Boundary conditions can depend on space and (optionally) time.
  - Time-dependent parameters (final time, time step, theta, initial condition) in a dedicated struct.
  - Quadrature order/type configurable.
  - Comprehensive validation with clear error reporting.
- **Usage**
  - User provides TOML configuration describing the PDE problem.
  - Executable reads config, sets up data, and runs correct solver.
  - Results are exported to output folder in both CSV and VTU format.

8. Results evaluation

In this section we present the validation and assessment of the developed FEM solver through a series of numerical experiments. Each test case has been designed to verify a specific aspect of the implementation: different spatial dimensions (1D, 2D, 3D), steady and time-dependent regimes, and a variety of boundary conditions (Dirichlet, Neumann, or mixed).
For every configuration we proceed as follows:

1. we specify the exact solution used in defining the problem. The forcing term and boundary conditions are derived using the exact solution. All the various coefficients (diffusion, transport, reaction) are non-zero.
2. we report the corresponding numerical solution, visualized in terms of scalar fields or line plots; it should closely resemble the exact result.
3. we explain why the obtained result is correct, comparing it either with the analytical solution (when available) or with the expected qualitative behaviour (e.g. linear profiles with constant Neumann data, symmetry of the solution, correct enforcement of boundary values).

All results detailed below can be reproduced using the corresponding Toml config file.
All the Toml files used below are in folder `config/good-examples/`
All our meshes can be found here. Place them in the `mesh/` folder.

8.1. Steady

8.1.1 1D

- TOML used: `sinusoidal_1d.toml`
- Exact solution: $u(x) = \sin(2\pi x)$

8.1.2 2D

- TOML used: `gaussian_bump_2d.toml`

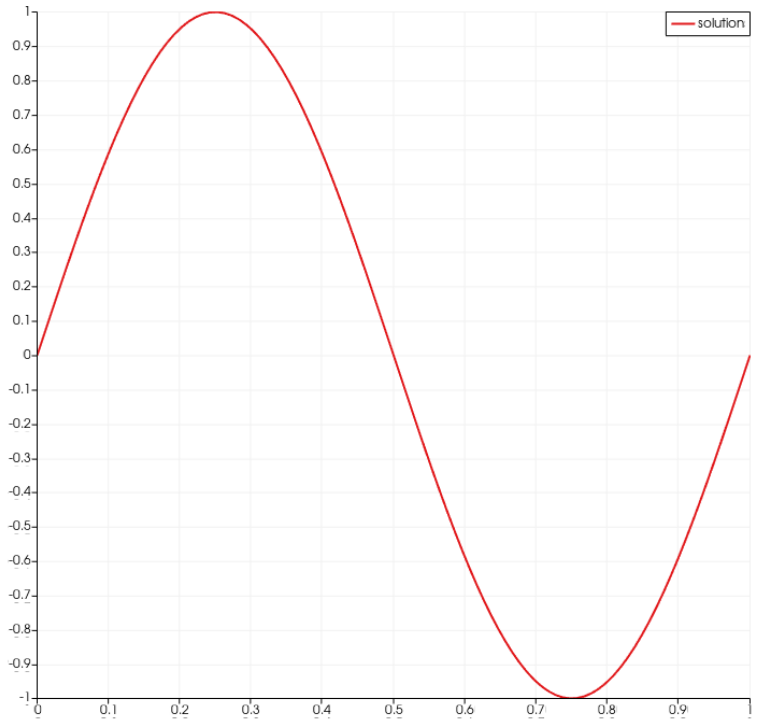

Figure 1: Exact solution for the 1D steady case.

- Exact solution: $u(x, y) = e^{-\frac{(x-0.5)^2+(y-0.5)^2}{0.1}}$

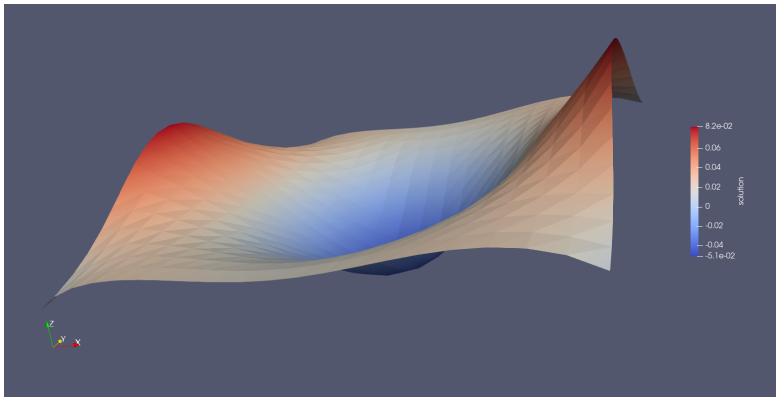

Figure 2: Exact solution for the 2D Gaussian bump (warped in 3^{rd} dimension).

- TOML used: `polynomial_2d.toml`
- Exact solution: $u(x, y) = x(1 - x)\, y(1 - y)$

8.1.3 3D

- TOML used: `gaussian_3d.toml`
- Exact solution: $u(x, y, z) = e^{-\frac{(x-0.5)^2+(y-0.5)^2+(z-0.5)^2}{0.02}}$

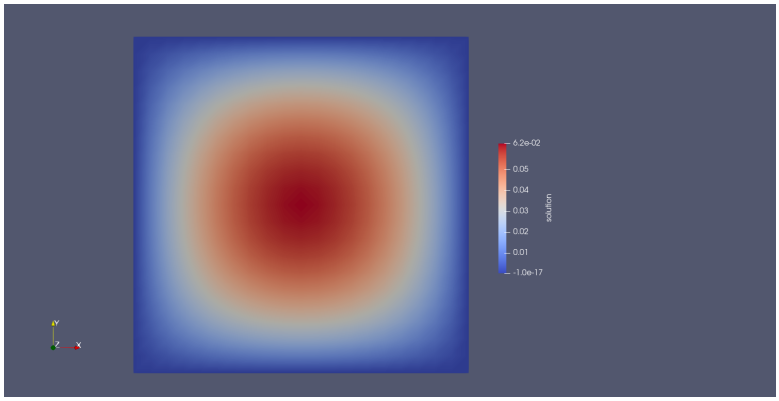

Figure 3: Exact solution for the 2D polynomial case.

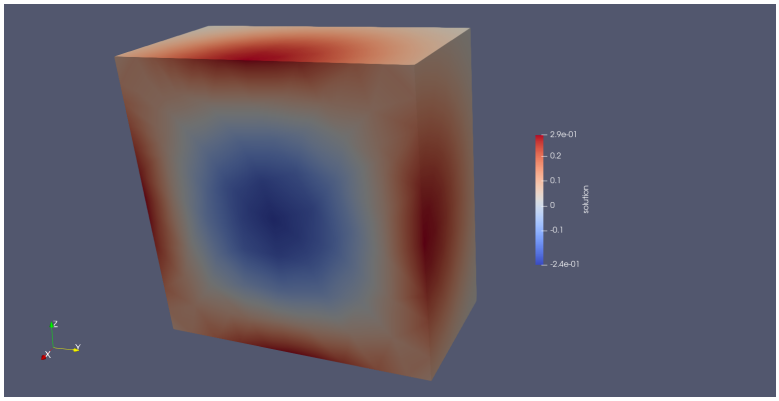

Figure 4: Exact solution for the 3D Gaussian bump.

8.2. Time-Dependent

Find videos of the time-dependent results here.

8.2.1 1D

- TOML used: `sinusoidal_1d_td.toml`
- Exact solution: $u(x,t) = \sin(2\pi x)\,\sin(t)$

8.2.2 2D

- TOML used: `wobble_2d_td.toml`
- Exact solution: $u(x,y,t) = \sin\big(2\pi(x+y)\big)\,\sin(2\pi t)$

8.2.3 3D

- TOML used: `polynomial_3d_td.toml`
- Exact solution: $u(x,y,z,t) = (x+y+z) * sin(2\pi\, t)$

8.3. Other results

Find all of our results here.

9. Performance evaluation

In addition to verifying the correctness of the solver, we also assessed its computational performance. The focus of this section is on the parallel efficiency achieved through the OpenMP implementation introduced in the assembly phase.
To quantify the benefits, we compared the execution time of the sequential version of the code against the parallel OpenMP version. Tests were conducted on three meshes of increasing size, in order to capture both the

overhead on small problems and the scalability on larger ones. For each case, we measured the wall-clock time as a function of the number of threads, and we report both the absolute timings and the relative speedup.

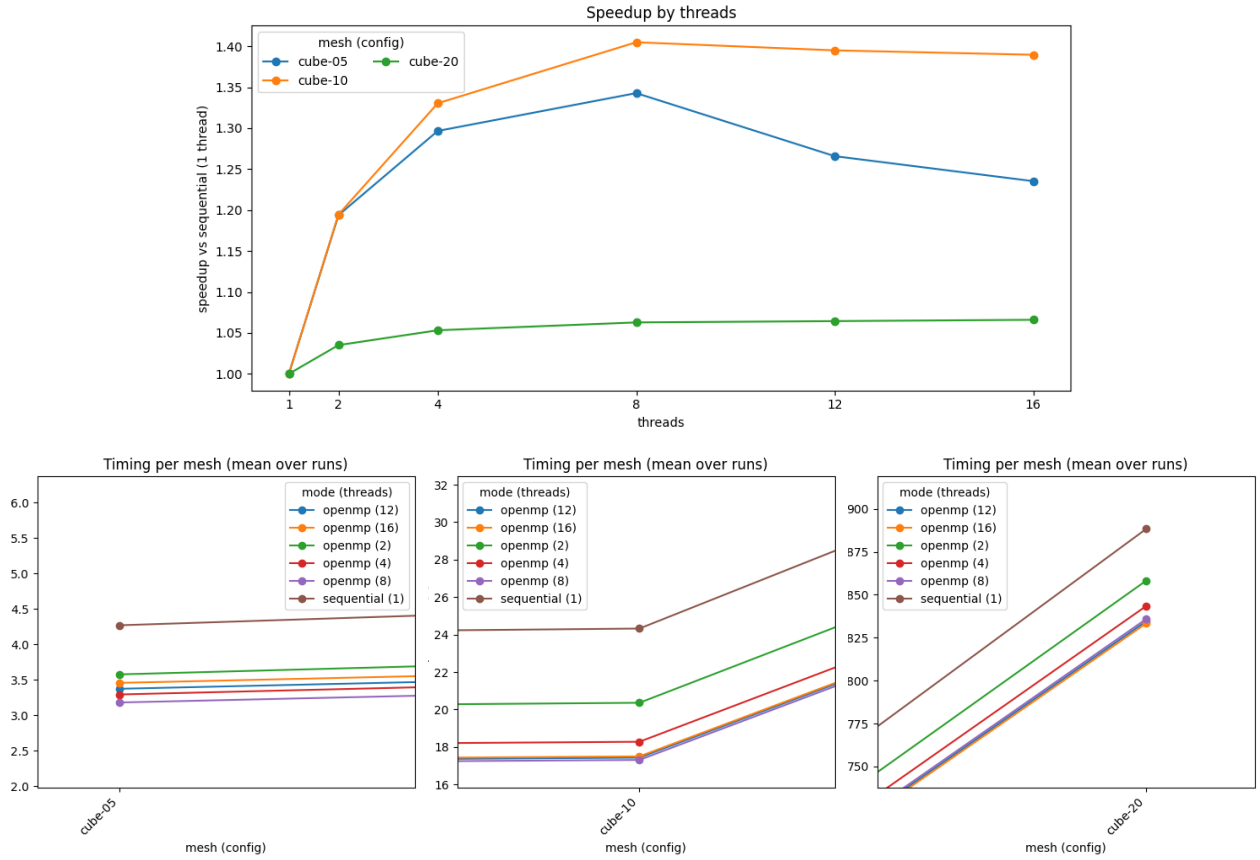

Figure 5: Performance evaluation: speedup (top) and execution times (bottom) across different meshes.

9.1. Comments

The performance experiments clearly highlight the impact of mesh size on the effectiveness of OpenMP parallelization. For the smallest mesh (`cube-5`), the parallel overhead dominates after 8 threads: the speedup remains limited and even decreases when increasing the number of threads, since the computational load is too small to compensate for thread management costs.

On the intermediate mesh (`cube-10`), we observe the best scalability: the assembly phase is large enough to benefit from parallelism, and the speedup reaches about 1.4× compared to the sequential run. Beyond 8 threads, the improvement tends to saturate due to increasing overhead costs.

For the largest mesh (`cube-20`), the execution time grows significantly, but the speedup remains close to 1.05× regardless of the number of threads. This indicates that in this configuration the parallel implementation is limited by several factors, but the main bottleneck is likely the final `solve()` step.

10. Conclusions

This work presented a modular, configuration-driven Finite Element Method solver for scalar diffusion-transport-reaction problems across one, two, and three dimensions, in both steady and time-dependent regimes. The architecture was designed to cleanly separate concerns: geometry and mesh handling, fields and quadrature, boundary conditions, assembly, linear algebra, and output are encapsulated behind dimension-generic interfaces. This separation enables reusability and progressive extension with minimal code duplication, while the TOML configuration layer decouples experiment design from implementation details, making runs reproducible and easy to compare. Practical performance on multi-core CPUs is achieved through OpenMP parallelism (also used internally by the Eigen library), and concise CSV/VTU exporters streamline post-processing and visualization in common scientific toolchains.

Beyond implementing the end-to-end workflow, the project contributes a uniform function abstraction for space- and time-dependent data, a consistent treatment of Dirichlet and Neumann conditions on boundaries of all

dimensions, and a quadrature layer that exposes a single interface to element and face integration. The resulting codebase is not only a working solver, but also a foundation on which alternative models, discretizations, and numerical experiments can be built rapidly.
Of course, there remain natural directions for improvement. Accuracy and robustness in advection-dominated regimes would benefit from stabilization techniques and higher-order elements; very large problems would profit from stronger preconditioning, multigrid strategies, and more advanced iterative methods; scalability could be extended through distributed-memory parallelism and GPU acceleration. Adaptive meshing and error-driven time-step control would further enhance efficiency and reliability. This FEM library provides a solid foundation on which to introduce such enchancements.

References

[1] T.J.R. Hughes. *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Courier Corporation, 2012.

[2] David A Dunavant. High degree efficient symmetrical gaussian quadrature rules for the triangle. *International Journal for Numerical Methods in Engineering*, 21(6):1129–1148, 1985. URL https://doi.org/10.1002/nme.1620210612.

[3] Tom Gustafsson and G. D. McBain. scikit-fem: A Python package for finite element assembly. *Journal of Open Source Software*, 5(52):2369, 2020. doi: 10.21105/joss.02369. URL https://github.com/kinnala/scikit-fem.