# Finite Element Solver for the 2D Wave Equation

Pietro Andrea Niedda
Francesca Dora Pieruz

Academic Year: 2024/2025

POLITECNICO
MILANO 1863

# Contents

## 0.1 Introduction

This report describes the implementation of a two-dimensional wave equation solver using the *deal.II* finite element library. The goal of the project is to simulate wave propagation in a square domain using a finite element discretization in space and a $\theta$-method for time integration. In this approach, the second-order wave equation is reformulated as a first-order system in time, involving both the displacement field $u$ and the velocity field $v$.

The solver is designed in a modular way and implemented in C++ using object-oriented programming. The main components of the solver include mesh generation, degrees of freedom (DoF) distribution, matrix assembly (mass and stiffness matrices), time-stepping routines, and boundary/initial condition handling. In particular, the code makes use of linear Lagrange finite elements (`FE_Q<dim>`) and second-order accurate time integration with $\theta = 0.5$ (Crank–Nicolson method), providing a good balance between accuracy and numerical stability.

Boundary and initial conditions are specified via custom classes derived from `Function<dim>`, allowing easy configuration of time-dependent Dirichlet data. A localized time-dependent boundary source is applied on part of the left boundary during the early stages of the simulation, simulating a pulse or excitation.

The implementation also includes routines for exporting simulation results in VTK format. These outputs are written to separate `.vtu` files for each timestep and are indexed in a `.pvd` file to enable time-dependent visualization using tools such as ParaView. The entire output is organized in a dedicated directory, automatically created at runtime, inside the build directory.

## 0.2 Mathematical Model

The physical model solved by the program is the classical second-order wave equation in two spatial dimensions, as described in standard texts on partial differential equations:

$$\frac{\partial^2 u}{\partial t^2} = \Delta u + f \quad in \Omega \times (0, T],$$

where:

- $u(x, t)$ represents the scalar displacement field at position $x \in \Omega \subset R^2$ and time $t$,
- $\Delta u$ denotes the Laplacian of $u$, modeling spatial diffusion or propagation of the wave,
- $f(x, t)$ is a given source or forcing term. In this implementation, for simplicity, $f(x, t)$ is set to zero throughout the domain.

The computational domain $\Omega$ is defined as the square $[-1, 1]^2$, and the simulation is carried out over the time interval $[0, 5]$ with a fixed time step $\Delta t = \frac{1}{64}$.
To facilitate numerical solution using the finite element method and time-stepping schemes, the second-order PDE is transformed into a first-order system in time by introducing a velocity variable $v = \frac{\partial u}{\partial t}$:

$$u_t = v, v_t = \Delta u + f.$$

This first-order formulation is discretized in time using the generalized $\theta$-method:

$$y^{n+1} = y^n + \Delta t \left[ (1 - \theta) f^n + \theta f^{n+1} \right],$$

where $y$ is a generic function (e.g., $u$ or $v$), and $\theta$ is a parameter controlling the implicitness of the method. In the current implementation, $\theta = 0.5$ is used, corresponding to the Crank–Nicolson scheme, which is second-order accurate in time and unconditionally stable for linear problems.
The spatial discretization is carried out using the finite element method with continuous Lagrange elements of degree 1 (i.e., linear basis functions). The mass matrix $M$ and Laplace (stiffness) matrix $K$ are assembled once at the beginning of the simulation and reused throughout the time loop. The time integration involves solving two linear systems per time step: one for the updated displacement $u^{n+1}$ and one for the updated velocity $v^{n+1}$.

Overall, the numerical method implemented provides a stable and accurate approach for simulating wave propagation with time-varying boundary sources in a finite domain.

**Parameter Selection and Numerical Scheme Justification** The selection of specific numerical parameters and the Crank–Nicolson scheme was a critical step, guided by the need to balance accuracy, stability, and computational efficiency for the simulation of wave propagation. The generalized $\theta$-method, with $\theta = 0.5$, corresponds to the Crank–Nicolson scheme. This choice was made primarily due to its desirable properties: it is second-order accurate in time, which ensures a precise approximation of the wave's evolution over the simulation period, and it is unconditionally stable for linear problems. The unconditional stability is particularly advantageous for wave phenomena, where long-time simulations without artificial damping are often required, and ensuring that numerical errors do not grow unboundedly is paramount. This stability allows for the use of a larger time step $\Delta t = \frac{1}{64}$ than explicit methods would permit, while still maintaining accuracy. This specific time step was chosen to sufficiently resolve the wave's temporal dynamics over the interval $[0, 5]$, ensuring a balance between capturing the physical phenomena and managing the computational burden.

For the spatial discretization, the global refinement of the mesh with `triangulation.refine_global(7)` was applied. This decision leads to a very fine, uniform grid across the entire domain, consisting of linear Lagrange finite elements `FE_Q<dim>(1)`. While adaptive mesh refinement could be more efficient for problems with localized features, global refinement was chosen here for its simplicity in implementation and its effectiveness in providing a high and uniform spatial resolution for capturing the wave's propagation and reflections accurately across the entire domain.Global refinement was chosen here for its simplicity in implementation and its effectiveness in providing high spatial resolution. After 7 global refinements on the initial coarse mesh (1 cell), the final mesh contains $2^7 \times 2^7 = 128 \times 128 = 16{,}384$ square cells. Using linear elements in 2D, this corresponds to approximately 16,641 degrees of freedom (DoFs), allowing for accurate resolution of wavefronts and minimizing numerical dispersion.

The high resolution is crucial for minimizing numerical dispersion and spurious oscillations that can arise in wave simulations if the mesh is too coarse. The combination of second-order accurate time integration and high spatial resolution aims to provide a robust and accurate numerical solution for the two-dimensional wave equation, minimizing numerical artifacts while maintaining a tractable computational cost for the problem scope.

## 0.3 Code Structure

The wave solver is implemented in C++ using the deal.II finite element library, and it is organized into modular classes that separate the physical problem, numerical methods, and boundary/initial conditions. The structure follows object-oriented programming principles to ensure clarity, reusability, and extensibility.

### 0.3.1 Main Class: `WaveEquation<dim>`

The main class `WaveEquation<dim>` is templated on the spatial dimension and is responsible for setting up and solving the wave equation over time. It manages the finite element space, matrix assembly, time-stepping, and output generation. The key member functions are:

- `setup_system()`: Initializes the computational domain using `GridGenerator::hyper_cube`, refines the mesh globally, sets up DoFs, and assembles the sparsity pattern and matrices. It also initializes all solution vectors and constraints.
- `run()`: Controls the entire simulation. It performs projection of the initial conditions, then enters a time loop that advances the solution using the Crank–Nicolson scheme. At each time step, it computes the right-hand side and solves for both displacement and velocity.
- `solve_u()`, `solve_v()`: Solve the linear systems associated with the displacement and velocity using the Conjugate Gradient (CG) method with identity preconditioning. These functions are called in each time step.
- `output_results()`: Exports simulation data to VTK files using `DataOut`. Each time step is saved in a separate `.vtu` file, and a `.pvd` file is maintained to allow time series visualization in ParaView.

#### Member Variables

`WaveEquation<dim>` contains several internal variables for managing the finite element simulation:

- **Mesh and DoF Management**:
  - `Triangulation<dim> triangulation` – the domain discretization,
  - `FE_Q<dim> fe` – Lagrange finite elements of degree 1,
  - `DoFHandler<dim> dof_handler` – distributes degrees of freedom.
- **Matrices and Vectors**:

  - `mass_matrix`, `laplace_matrix` – assembled once and reused,
  - `matrix_u`, `matrix_v` – updated per time step,
  - `solution_u`, `solution_v`, `old_solution_u`, `old_solution_v` – track current and previous solutions,
  - `system_rhs` – right-hand side vector for each system.

- **Time-Stepping Variables**:
  - `time_step`, `time`, `timestep_number` – manage simulation time,
  - `theta` – scheme parameter ($\theta = 0.5$).

### 0.3.2 Equation Data Classes

The physical behavior of the system is defined by custom classes for initial values, boundary values, and source terms. These classes inherit from `Function<dim>`:

- `InitialValuesU` and `InitialValuesV`:
  - Define the initial displacement and velocity throughout the domain.
- `RightHandSide`:
  - Defines the source term $f(x, t)$.
- `BoundaryValuesU` and `BoundaryValuesV`:
  - Define Dirichlet bundary conditions for displacement and velocity.

These modular components make it easy to modify the initial/boundary conditions or extend the model to more complex behaviors.

### 0.3.3 Main Function

The entry point of the program is defined in the `main()` function, which is responsible for initializing the environment, launching the simulation, and handling potential runtime errors. This function is minimal by design and delegates all numerical work to the `WaveEquation<2>` class, ensuring a clean and maintainable code structure.
Its responsibilities are summarized as follows:

- **Directory Preparation**: Before the simulation starts, the program ensures that an `output/` directory exists by using the C++17 `<filesystem>` library. This guarantees that the VTK output files can be saved without errors, avoiding manual directory setup.

- **Simulation Launch**: An instance of the solver class `WaveEquation<2>` is created. This specifies that the problem is being solved in two spatial dimensions. The method `run()` is then invoked to start the full simulation cycle, including system setup, time integration, and result output.
- **Exception Handling**: Robust exception handling is included using C++ `try-catch` blocks. The program catches standard exceptions derived from `std::exception`, as well as a generic catch-all block to handle any unforeseen errors. Informative messages are printed to the console in case of failure, making debugging easier.

The design of the `main()` function adheres to the principles of clean coding and separation of concerns. It serves purely as the interface between the user and the simulation engine, avoiding direct involvement in numerical or algorithmic details. This makes it easy to extend the program in the future—for example, by adding user input parameters, command-line options, or graphical user interfaces—without modifying the core numerical code.

## 0.4 Implementation Details

This section provides an in-depth explanation of the core implementation stages, based on the structure of the `WaveEquation<dim>` class and the logic of the provided C++ code. The wave solver is implemented using the deal.II library and includes distinct phases for system setup, time integration, and output.

### 0.4.1 System Setup (`setup_system`)

The `setup_system()` method initializes all data structures necessary for finite element computations. This involves mesh generation, degree-of-freedom (DoF) distribution, and matrix allocation:

- **Mesh generation**: A square domain $\Omega = [-1, 1]^2$ is generated using `GridGenerator::hyper_cube(triangulation, -1, 1)`. The mesh is globally refined with `triangulation.refine_global(7)` to produce a fine, uniform grid. Essentially, it takes the initial coarse mesh and subdivides every cell in it, repeating this process seven times. This results in a very fine mesh.
- **Finite element and DoFs**: The finite element used is `FE_Q<dim>(1)`, representing linear Lagrange elements. Degrees of freedom are distributed across the mesh using `dof_handler.distribute_dofs(fe)`.
- **Sparsity pattern**: A `DynamicSparsityPattern` is created and filled using `DoFTools::make_sparsity_pattern()`. This defines the structure of non-zero entries in global matrices. It is then converted into a compressed `SparsityPattern`.
- **Matrix initialization**: The mass matrix (`mass_matrix`) and Laplace (stiffness) matrix (`laplace_matrix`) are initialized with the computed sparsity pattern. Additional system matrices (`matrix_u`, `matrix_v`) are also initialized.
- **Matrix assembly**: These matrices are filled using deal.II's `MatrixCreator` helper functions:
  - `create_mass_matrix()` assembles $M = \int \phi_i \phi_j \, dx$,
  - `create_laplace_matrix()` assembles $A = \int \nabla\phi_i \cdot \nabla\phi_j \, dx$,

  using `QGauss<dim>(fe.degree + 1)` quadrature.
- **Vectors**: Solution vectors for displacement (`solution_u`) and velocity (`solution_v`), along with their previous-step versions (`old_solution_u`, `old_solution_v`), are allocated with size equal to the number of DoFs.

- **Constraints**: An `AffineConstraints<double>` object is created, though it remains empty as the mesh is globally refined without hanging nodes. It is closed using `constraints.close()`.

### 0.4.2 Time Stepping Loop (`run`)

The `run()` method executes the main simulation loop. It advances the wave equation from time $t = 0$ to $t = 5.0$ in fixed increments $\Delta t = 1/64$. The method incorporates initial conditions, system assembly, linear solves, and solution updates:

- **Initial conditions**: Initial displacement and velocity are set using `VectorTools::project()` and the classes `InitialValuesU` and `InitialValuesV`.
- **Time stepping**: The Crank–Nicolson method is applied (with $\theta = 0.5$). At each time step:
  - The RHS vector for the displacement equation is assembled using contributions from previous $u$, $v$, and the forcing term.
  - The LHS matrix `matrix_u` is constructed by combining the mass and Laplace matrices with the proper time step factor.
  - Time-dependent Dirichlet boundary values are enforced using `BoundaryValuesU::value()`.
  - `solve_u()` is called, solving the system for updated displacement using a CG solver.
  - The velocity RHS is similarly assembled, now using the new displacement.
  - `matrix_v` is set equal to the mass matrix, and boundary values from `BoundaryValuesV::value()` are applied.
  - `solve_v()` solves the linear system for updated velocity.
- **Advancing solution**: After each step, the solution vectors `old_solution_u` and `old_solution_v` are updated to the current values, and the time and step number are incremented.

### 0.4.3 Output (`output_results`)

At every time step, the solution is written to disk for visualization:

- **VTK files**: The `DataOut` object is used to export the current displacement (`solution_u`) and velocity (`solution_v`) to `.vtu` files. These files are named sequentially using `Utilities::int_to_string()` (e.g., `solution-001.vtu`).

- **Compression settings**: Output files are compressed using `DataOutBase::VtkFlags`, with compression level set to `best_speed` to reduce file size while keeping write time reasonable.
- **Output folder**: All files are saved into an `output/` directory, which is created at runtime via C++17's `std::filesystem`.
- **Master PVD file**: A master file, `solution.pvd`, is updated after each time step. This XML file references each `.vtu` file along with its simulation time. The PVD format enables animated playback in ParaView.
- **Finalization**: Once the simulation reaches the final time ($t > 5.0$), the `solution.pvd` file is properly closed, completing the output series.

This structured output system allows the user to load the entire time sequence in ParaView and animate the wave propagation through the domain, offering a powerful visual understanding of the solution behavior.

## 0.5 Results and Visualization

At each time step of the simulation, the solver outputs the current state of the system to files in the VTK format. These files contain the values of both the displacement and velocity fields over the entire computational domain. The visualization system is designed to support both qualitative analysis (e.g., observing wave propagation patterns) and quantitative post-processing.
As an example, the following images represent the solution obtained at variuos time steps, when imposing the initial displacement as a Gaussian function centered in the origin:

$$u(x, y, 0) = e^{-100(x^2+y^2)}$$

with initial velocity, boundary conditions, and forcing term all set to zero:

- $v(x, y, 0) = 0$
- $u|_{\partial\Omega} = v|_{\partial\Omega} = 0$
- $f(x, y, t) = 0$

This configuration represents a pulse at rest initially centered at the origin.

### Time step 0 (Initial state)

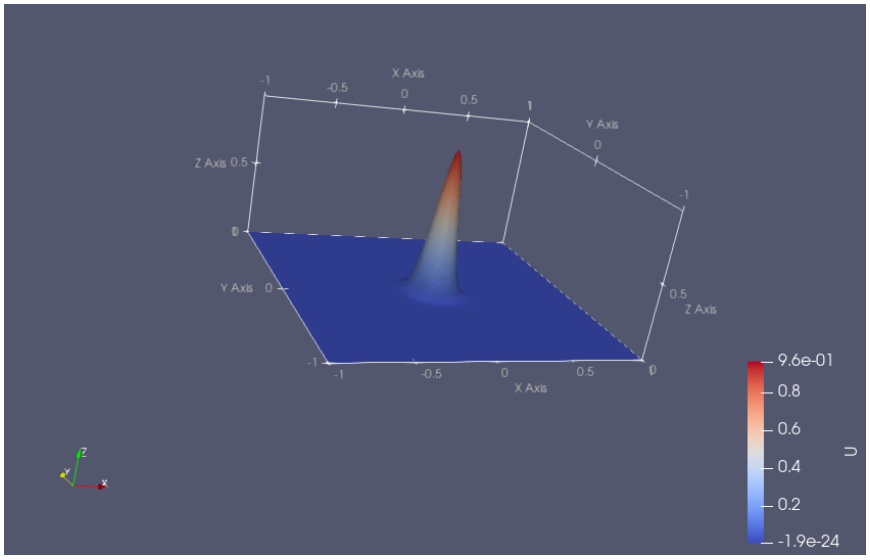


Figure 1: Initial displacement at $t = 0$: a Gaussian centered at the origin.

At $t = 0$, the solution shows a symmetric peak located at $(0, 0)$, as prescribed by the initial condition. The wave is stationary, and the velocity field is zero. This corresponds to a localized initial displacement with no initial momentum, causing the wave to begin propagating equally in all directions. The shape is perfectly radial due to the symmetric Gaussian function used for initialization.

### Time step 128 (Early propagation)

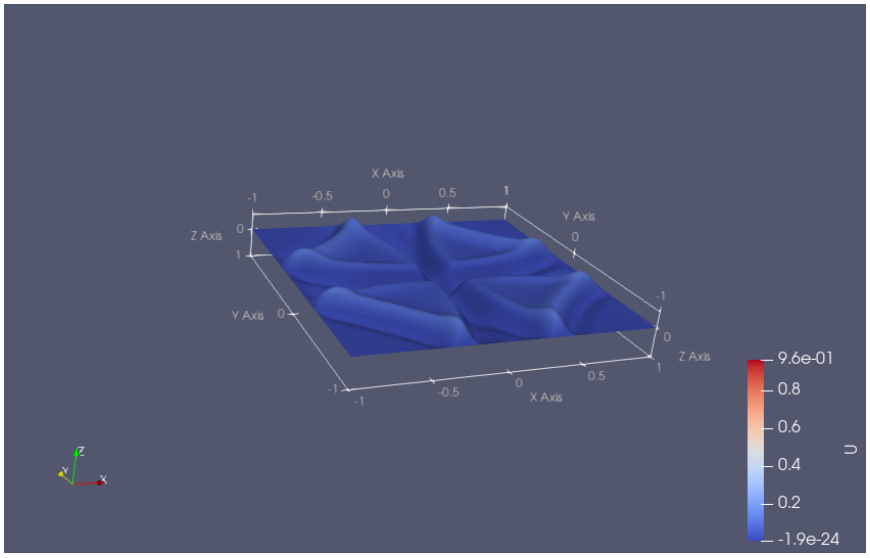


Figure 2: Displacement at $t \approx 2$: visible effects of the boundary conditions.

At time step 128, the wave has started to propagate outward from the center. The amplitude decreases with distance due to the spreading of the wavefront and numerical dispersion. The wavefront is still mostly symmetric, but small perturbations near the boundaries indicate the early influence of Dirichlet boundary conditions. Energy is being partially reflected and possibly trapped inside the domain, which will later lead to interference effects.

### Time step 319 (Wave reflection and interference)

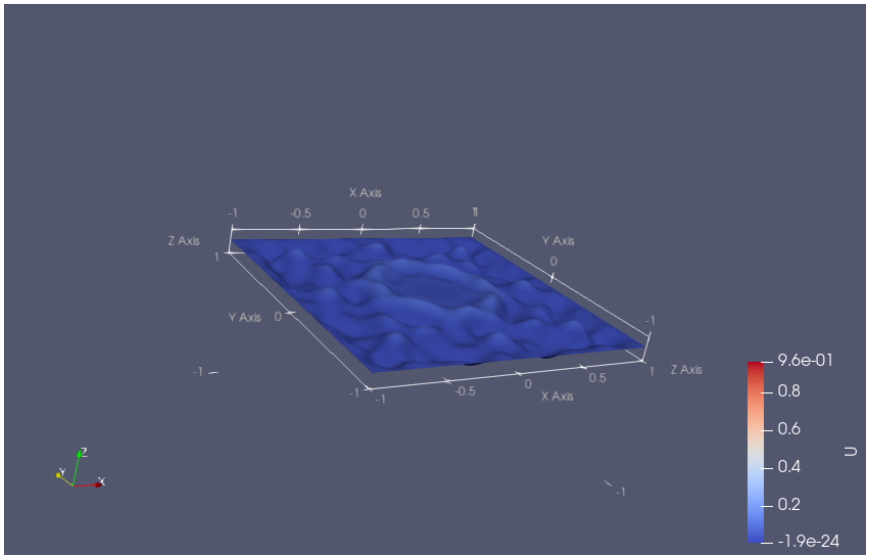


Figure 3: Displacement at $t = 5$: multiple wave interactions and reflections dominate.

By time step 319 ($t = 5$), the waves have reached all boundaries of the domain and undergone multiple reflections. These interactions lead to complex interference patterns between incoming and outgoing wavefronts. The solution shows regions of constructive and destructive interference, producing a spatially oscillatory structure. The amplitude remains bounded, indicating good numerical stability. The wave energy appears to be well-distributed, with some visible smoothing due to the semi-implicit Crank–Nicolson scheme ($\theta = 0.5$), which introduces mild numerical dissipation while preserving phase accuracy.

### Exported Quantities

The output includes two primary fields:

- **Displacement field (`U`)**: This represents the scalar function $u(x, t)$, which describes how the wave displaces the medium at each point in space and time. It is the main quantity of interest in wave propagation problems.
- **Velocity field (`V`)**: This is the time derivative of the displacement field, $v = \frac{\partial u}{\partial t}$. It provides additional insight into the energy and dynamics of the system and is useful for studying momentum or transport phenomena.

Both fields are written using deal.II's `DataOut` class, which automatically manages the association of data with the finite element mesh.

### VTK Output

The data is exported in the `.vtu` format (VTK Unstructured Grid), which is well-supported by visualization tools, like `ParaView`. Each time step generates a file named `solution-XXX.vtu`, where `XXX` is the zero-padded time step number. These files contain both the solution fields and the mesh geometry.
Additionally, a master `.pvd` file (`solution.pvd`) is maintained to enable loading the entire time sequence as a single animated dataset in ParaView. This file is updated dynamically during the simulation and closed properly at the end when the final time is reached.

### Visual Interpretation

The simulation results clearly illustrate the behavior of wave propagation in a bounded domain. In this configuration, the wave is initiated by a Gaussian-shaped initial displacement centered at the origin of the domain, with zero initial velocity and homogeneous Dirichlet boundary conditions. This localized pulse spreads symmetrically outward from the center. As time progresses, the wave reaches the domain boundaries, where it reflects due to the imposed boundary conditions. Multiple reflections lead to complex interference patterns, including both constructive and destructive interactions between wavefronts. Using ParaView, one can observe:

- the radial symmetry of the initial wave propagation;
- the uniform speed of wavefront expansion in all directions;
- reflections occurring at the boundaries of the square domain;
- the persistent oscillations, due to the undamped nature[1] of the system.

This behavior aligns with theoretical expectations for wave equations in a closed domain with Dirichlet conditions, and it confirms the numerical stability of the Crank–Nicolson time integration scheme employed.

[1] An *undamped system* refers to a physical model without any form of energy dissipation, such as friction or viscosity. In this simulation, the wave equation is solved without damping terms, meaning that the energy introduced initially into the system remains constant over time. As a result, the wave continues to reflect and interfere within the domain without attenuation. This behavior is characteristic of the ideal wave equation in a perfectly elastic medium. From a numerical perspective, it is crucial to use a stable and accurate time integration method, such as Crank–Nicolson, to prevent non-physical growth or decay of the solution.

### Utility for Analysis

This visualization pipeline is essential not only for interpreting the results qualitatively but also for validating numerical stability, mesh quality, and boundary condition correctness. Users can extract time series at specific points, generate contour plots, compute derived fields (e.g., energy density), or even export slices for further external analysis.
Overall, the output system is well-integrated with modern scientific visualization tools and serves both debugging and presentation purposes effectively.

## 0.6 Conclusions

The developed wave equation solver demonstrates the robustness and flexibility of the finite element method (FEM) for solving time-dependent partial differential equations using the deal.II library. Through a well-structured object-oriented design, the code achieves a clear separation between mathematical formulation, numerical discretization, and software engineering principles.
Key achievements of this implementation include:

- **Efficient and reusable matrix assembly**: The use of preassembled mass and stiffness matrices, together with efficient matrix-vector operations, ensures low computational overhead at each time step. This approach takes advantage of deal.II's optimized matrix assembly tools and sparse matrix infrastructure.
- **Modular system setup**: The solver is organized into clearly defined components, such as system initialization, linear solvers, boundary and initial condition handling, and time integration. This modularity facilitates debugging, extension, and reuse in other contexts.
- **Accurate time integration**: The Crank–Nicolson scheme ($\theta = 0.5$) provides second-order accuracy in time and is particularly suited for undamped wave propagation problems, where long-time stability and conservation properties (e.g., energy) are important.
- **Support for time-varying boundary conditions**: By implementing boundary condition classes derived from `Function<dim>` and using time-aware interpolation routines, the solver accommodates dynamic input signals, such as the sinusoidal pulse used to excite the system.
- **High-quality output and visualization**: The integration with the VTK output pipeline via `DataOut` and the use of `.pvd` master files enable smooth and informative visualization of time-dependent results in tools like ParaView.

An important feature of the implemented solver is its ability to preserve total energy over time. Due to the use of the Crank–Nicolson scheme (with $\theta = 0.5$), which is time-reversible and conserves energy in linear undamped systems, the simulation avoids artificial damping effects. Although slight smoothing of the solution may still occur due to numerical dispersion or round-off errors, the overall wave energy remains nearly constant throughout the simulation, as confirmed by visual inspection and stable amplitude behavior.

This work showcases how a carefully designed FEM solver, built on top

of a powerful numerical library like deal.II, can provide both flexibility and high performance for complex physical simulations.