POLITECNICO DI MILANO

INHOMOGENEOUS, VISCOUS BURGERS EQUATION SOLVER

ADVANCED PROGRAMMING FOR SCIENTIFIC COMPUTING, A.Y. 2023-2024

Student: Francesco Pettenon

Supervisor: Prof. De Falco Carlo

Teacher: Prof. Formaggia Luca

5th September 2024

TABLE OF CONTENTS

Contents

1 Introduction

The paper "Efficient Quantum Algorithm for Dissipative Nonlinear Differential Equations" [1] presents a groundbreaking quantum algorithm designed to efficiently address the complexities of solving dissipative nonlinear differential equations. These equations are notoriously difficult to solve due to their inherent nonlinearity and dissipation. By utilizing Carleman linearization, the algorithm transforms nonlinear equations into a linear form that is more tractable for quantum computation. The key advancement of this algorithm lies in its ability to provide an exponential increase in computational efficiency compared to traditional methods, particularly when dissipation dominates over nonlinearity and external forcing terms. This innovative approach holds significant promise for applications in diverse fields such as fluid dynamics and epidemiology.

In this project, we implement this quantum algorithm using C++ to solve dissipative nonlinear differential equations. Our evaluation involves comparing the algorithm's performance against conventional methods typically employed for solving partial differential equations (PDEs) and ordinary differential equations (ODEs). These classical methods include numerical techniques like finite difference methods and Runge-Kutta schemes. The primary objective is to assess the accuracy, computational efficiency, and potential benefits of the quantum approach in tackling complex differential equations.

Additionally, the project employs Git for version control to ensure reproducibility and repeatability, key factors in scientific computing. By following the instructions provided, the results can be replicated on any machine without discrepancies. The implementation leverages software engineering techniques and skills, such as automated dependency management through a Makefile, Doxygen for documentation generation, and optimized memory management practices, including automated library loading and memory cleaning.

Specifically, we begin by solving a one-dimensional Burgers' equation, a canonical PDE, by transforming it into a linear form suitable for applying the quantum algorithm. The report is organized as follows: Section 2 provides a brief overview of the mathematical models and methods employed. Section 3 details the implementation in C++, focusing on the design choices that underpin the entire computational pipeline, from parameter input and function definitions to the management of solutions and matrices. Section 4 presents the simulation parameters and the results obtained. Finally, Section 5 offers conclusions and suggests potential improvements for future work, both in terms of implementation and modeling.

The source code is available in this repository.

2 Models

2.1 Burgers equation

We consider the celebrated Navier–Stokes equation

$$\partial_t \mathbf{u} + (\mathbf{u} \cdot \nabla)\mathbf{u} + \beta \mathbf{u} = \nu \nabla^2 \mathbf{u} + \mathbf{f} \tag{1}$$

with linear damping and a forcing term. Equations of the form 1 are ubiquitous in fluid mechanics [2], and related models such as those studied in refs. [3], [4], [5] are used to describe the formation of large-scale structure in the universe. Similar equations also appear in magnetohydrodynamics [6] and in models that describe the motion of free particles that stick to each other upon collision [7]. In the inviscid case, $\nu = 0$, the resulting Euler type equations with linear damping are also of interest, both for modeling micromechanical devices [8] and for their intimate connection with viscous models [9].

As a specific example, we consider the one-dimensional forced viscous Burgers equation

$$\partial_t u + u\partial_x u = \nu \partial_x^2 u + f \tag{2}$$

which is the one-dimensional case of Eq. 1 with $\beta = 0$. This is often used as a simple model of convective flow [10]. For concreteness, let the initial condition be

$$u(x, 0) = U_0 sin(\frac{2\pi x}{L_0})$$

on the domain $x \in \left[-\frac{L_0}{2}, \frac{L_0}{2}\right]$, and use Dirichlet boundary conditions $u(-\frac{L_0}{2}, 0) = u(\frac{L_0}{2}, 0) = 0$. We force this equation using a localized off-center Gaussian with a sinusoidal time dependence, given by

$$f(x, t) = U_0 \exp(-\frac{(x - L_0/4)^2}{2(L_0/32)^2}) cos(2\pi t)$$

.

2.2 Space Discretization

To solve the one-dimensional forced viscous Burgers equation using the Carleman method, we first discretize the spatial domain into n_x points and apply central differences for the derivatives. This transforms the partial differential equation into a system of ordinary differential equations (ODEs). Let's walk through the steps in detail.

Conversion to System of ODEs

Given the Burgers equation 2, we discretize the spatial domain x into n_x points, so that $x_j = j\Delta x$ where Δx is the spatial step size and j ranges from 1 to n_x. Let $u_j(t)$ denote the approximation of $u(x_j, t)$ at the grid point x_j and time t. Using central difference approximation:

- **First Derivative (Advection Term $\partial_x u$):**

$$\partial_x u_j \approx \frac{u_{j+1} - u_{j-1}}{2\Delta x}$$

- **Second Derivative (Diffusion Term $\partial_x^2 u$):**

$$\partial_x^2 u_j \approx \frac{u_{j+1} - 2u_j + u_{j-1}}{(\Delta x)^2}$$

Using these approximations, the discretized form of the Burgers equation at each grid point j is:

$$\frac{du_j}{dt} + u_j \frac{u_{j+1} - u_{j-1}}{2\Delta x} = \nu \frac{u_{j+1} - 2u_j + u_{j-1}}{(\Delta x)^2} + f_j(t),$$

where $f_j(t)$ represents the value of the external forcing function at point x_j.

The discretized equation can be rewritten in the following form:

$$\frac{du_j}{dt} = -\frac{u_j}{2\Delta x}(u_{j+1} - u_{j-1}) + \nu \frac{u_{j+1} - 2u_j + u_{j-1}}{(\Delta x)^2} + f_j(t). \tag{3}$$

This is a system of ODEs for $u = [u_1, u_2, \ldots, u_{n_x}]^T$.

Carleman Linearization and Quadratic Form

In the Carleman linearization method, the goal is to rewrite the system of ODEs into a quadratic form:

$$\frac{du}{dt} = F_2 u^{\otimes 2} + F_1 u + F_0(t), \tag{4}$$

where:

- $u \in \mathbb{R}^{n_x}$ is the vector of solutions $[u_1(t), u_2(t), \dots, u_{n_x}(t)]^T$.
- $u^{\otimes 2} \in \mathbb{R}^{n_x^2}$ is the Kronecker product, representing all quadratic combinations of u_j, specifically:

$$u^{\otimes 2} = [u_1^2, u_1 u_2, \dots, u_1 u_{n_x}, u_2 u_1, \dots, u_{n_x} u_{n_x - 1}, u_{n_x}^2]^T.$$

Composition of Matrices $F_0(t)$, F_1, and F_2

- **Matrix F_2:**
  - This matrix captures the quadratic interactions between the components of u.
  - Since the advection term $u_j \partial_x u$ introduces nonlinearities (quadratic terms) in the equation, F_2 will primarily represent these contributions.
  - The matrix is sparse and mostly composed of zeros, except for the entries corresponding to interactions $u_j u_{j+1}$ and $u_j u_{j-1}$ for the advection term.

$$F_2 = -\frac{1}{2\Delta x} \begin{bmatrix} 0 & I & 0 & \cdots & 0 \\ -I & 0 & I & \cdots & 0 \\ 0 & -I & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix}_{n_x \times n_x^2},$$

  where I is the identity matrix of appropriate dimensions. The nonzero blocks in F_2 correspond to the interaction terms $u_j u_{j+1}$ (represented by I and $-I$) and are placed in such a way that they capture the product $u_j(u_{j+1} - u_{j-1})$ from the advection term.

- **Matrix F_1:**
  - F_1 contains the linear coefficients corresponding to the linear parts of the equation, such as the diffusion term $\nu \partial_x^2 u$.
  - It is also a sparse matrix with entries that represent the central difference approximation for the second derivative.

  Given the central difference discretization, F_1 is a tridiagonal matrix:

$$F_1 = \nu \frac{1}{(\Delta x)^2} \begin{bmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & \cdots & 0 \\ 0 & 1 & -2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & -2 \end{bmatrix}_{n_x \times n_x}.$$

  Here, the diagonal entries -2 correspond to the $-2u_j$ terms, while the off-diagonal entries 1 correspond to the u_{j+1} and u_{j-1} terms.

- **Vector $F_0(t)$:**
  - $F_0(t)$ accounts for the external forcing term $f_j(t)$ and any time-dependent contributions.
  - This vector varies with time and directly adds to the system without any dependence on u.

2.3 Carleman Number

Before proceeding with the algorithm, it is crucial to evaluate the Carleman number R, which functions similarly to the Reynolds number in fluid dynamics. The Carleman number plays an essential role in convergence estimations. Specifically, we require $R < 1$ as a key hypothesis for the convergence theorem.

Given that the eigenvalues of F_1 satisfy the condition $\text{Re}(\lambda_n) \leq \cdots \leq \text{Re}(\lambda_1) < 0$, we define the Carleman number as:

$$R = \frac{1}{|\text{Re}(\lambda_1)|}\left(\|u_{\text{in}}\|\|F_2\| + \frac{\|F_0\|}{\|u_{\text{in}}\|}\right).$$

This formulation of R provides a scalar measure that is essential for assessing the conditions under which the algorithm converges. For a detailed discussion of the theoretical background and related theorems, we refer to [1].

2.4 Carleman Procedure

Carleman linearization is a method for converting a finite-dimensional system of nonlinear differential equations into an infinite-dimensional linear system. This transformation is achieved by introducing powers of the variables into the system, allowing it to be written as an infinite sequence of coupled linear differential equations. The system is then truncated to N equations, where the truncation level N depends on the allowed approximation error, resulting in a finite linear ODE system.

Given a system of quadratic ODEs 4, we apply the Carleman procedure to obtain the system of linear ODEs, where we denote A as Carleman Matrix

$$\frac{d\hat{y}}{dt} = A(t)\hat{y} + b(t), \quad \hat{y}(0) = \hat{y}_{\text{in}}, \tag{5}$$

with the tridiagonal block structure:

$$\frac{d}{dt}\begin{pmatrix}\hat{y}_1\\ \hat{y}_2\\ \vdots\\ \hat{y}_N\end{pmatrix} = \begin{pmatrix} A_1^1 & A_1^2 & 0 & \cdots & 0\\ A_2^1 & A_2^2 & A_2^3 & \cdots & 0\\ 0 & A_3^2 & A_3^3 & \cdots & 0\\ \vdots & \vdots & \vdots & \ddots & A_{N-1}^N\\ 0 & 0 & 0 & A_N^{N-1} & A_N^N\end{pmatrix}\begin{pmatrix}\hat{y}_1\\ \hat{y}_2\\ \vdots\\ \hat{y}_N\end{pmatrix} + \begin{pmatrix}F_0(t)\\ 0\\ \vdots\\ 0\end{pmatrix},$$

where $\hat{y}_j = u^{\otimes j} \in \mathbb{R}^{n^j}$, and $\hat{y}_{\text{in}} = [u_{\text{in}}; u_{\text{in}}^{\otimes 2}; \ldots; u_{\text{in}}^{\otimes N}]$. The matrices $A_j^{j+1} \in \mathbb{R}^{n^j \times n^{j+1}}$, $A_j^j \in \mathbb{R}^{n^j\times n^j}$, and $A_j^{j-1} \in \mathbb{R}^{n^j \times n^{j-1}}$ for $j \in [N]$ are defined as:

$$\begin{aligned} A_j^{j+1} &= F_2 \otimes I^{\otimes(j-1)} + I \otimes F_2 \otimes I^{\otimes(j-2)} + \cdots + I^{\otimes(j-1)} \otimes F_2,\\ A_j^j &= F_1 \otimes I^{\otimes(j-1)} + I \otimes F_1 \otimes I^{\otimes(j-2)} + \cdots + I^{\otimes(j-1)} \otimes F_1,\\ A_j^{j-1} &= F_0(t) \otimes I^{\otimes(j-1)} + I \otimes F_0(t) \otimes I^{\otimes(j-2)} + \cdots + I^{\otimes(j-1)} \otimes F_0(t).\end{aligned} \tag{6}$$

2.5 Time discretization

To construct a system of linear equations, we divide the interval $[0, T]$ into $m = \frac{T}{h}$ time steps and apply the forward Euler method to equation 5, yielding

$$y_{k+1} = [I + A(kh)h]\, y_k + b(kh), \quad \text{for } k \in [m+1]_0 := \{0, 1, \ldots, m\},$$

where $y_k \in \mathbb{R}^{\Delta}$ approximates $\hat{y}(kh)$ at each time step k, with the initial condition $y_0 = y_{\text{in}} := \hat{y}(0) = \hat{y}_{\text{in}}$.

2.6 QLSA Algorithm

Additionally, we assume that all y_k remain equal for $k \in [m+p+1]_0 \setminus [m+1]_0$, where p is a sufficiently large integer.

This procedure results in an $(m+p+1)\Delta \times (m+p+1)\Delta$ linear system:

$$L|Y\rangle = |B\rangle,$$

which encodes the previous equation and provides a numerical solution at time T, where

$$L = \sum_{k=0}^{m+p} |k\rangle\langle k| \otimes I - \sum_{k=1}^{m} |k\rangle\langle k-1| \otimes [I + A((k-1)h)h] - \sum_{h=m+1}^{m+p} |k\rangle\langle k-1| \otimes I,$$

and

$$|B\rangle = \frac{1}{\sqrt{B_m}} \left(|y_{\text{in}}\rangle\langle 0| \otimes |y_{\text{in}}\rangle + \sum_{k=1}^{m} |b((k-1)h)\rangle\langle k| \otimes |b((k-1)h)\rangle \right),$$

with a normalizing factor B_m. Observe that the system has a lower triangular structure:

$$\begin{pmatrix} I & & & & & & & \\ -[I + A(0)h] & I & & & & & & \\ & \ddots & \ddots & \ddots & & & & \\ & & -[I + A((m-1)h)h] & I & & & & \\ & & & -I & I & & & \\ & & & & \ddots & \ddots & \ddots & \\ & & & & & -I & I & \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_m \\ y_{m+1} \\ \vdots \\ y_{m+p} \end{pmatrix} = \begin{pmatrix} y_{\text{in}} \\ b(0) \\ \vdots \\ b((m-1)h) \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

Once the solutions y_k have been computed for all time steps k, we proceed to filter the solution to obtain $\hat{y}_1$. This step allows us to subsequently derive the final approximated solution $u_{\text{c}N}(x, t)$ for each $N = 1, \ldots, N_{\text{Max}}$ (in our case, $N_{\text{Max}} = 4$).

The filtered solution (for each N) is stored in a matrix. For each time step k from 0 to $nt - 1$, we extract the first n_x components of y_k.

This process is repeated for each N, providing the final approximated solution $u_{\text{c}N}(x, t)$.

2.7 Other Classical Methods

To compare the solution obtained using the Carleman method, we also implement the following numerical methods for further analysis. Starting with the space-discretized version of the problem, we apply both the Direct Euler Method and the RK45 method to solve the system [11]. Additionally, we directly solve the partial differential equation (PDE) using a central difference scheme [12], effectively reproducing the functionality of MATLAB's `pdepe` solver for numerical integration.

By applying these methods, we ensure a robust comparison of the numerical solutions across different approaches and highlight the strengths and weaknesses of each.

3 Implementation in C++

The implementation of the Inhomogeneous, Viscous Burgers Equation Solver is structured to ensure modularity, flexibility, and efficiency. The project is organized into several components, each responsible for different aspects of the simulation, from parameter management to numerical solving and plotting. The design follows a class-based approach with a clear separation of concerns, allowing for easy maintenance, extension, and testing.

3.1 Project Structure and Design Choices

The project is organized into the following main components:

- **Main Program (`main.cpp`):** This is the entry point of the program, responsible for setting up and running the simulations. It reads parameters from external configuration files (`.pot` or `.json`) using the `GetPot` and `nlohmann/json` libraries, respectively. It then initializes the simulation environment and invokes the appropriate solver and other utilities.

- **Simulation Management (`src/MainSimulation.cpp`, `src/MainSimulation.hpp`):** This component encapsulates the core functionality required to initialize and run simulations. The `MainSimulation` class serves as the base class, providing methods for initializing simulation parameters, managing the simulation loop, and interfacing with solvers.

- **Parameter Handling (`src/params/`):** This directory contains classes responsible for reading and managing simulation parameters. The use of the `GetPot` and `nlohmann/json` libraries ensures flexibility in the input format, allowing users to easily configure simulations.

- **Discretization (`src/discretization/`):** This module handles the spatial and temporal discretization of the equations. Different discretization schemes can be implemented and easily switched depending on the simulation requirements.

- **Initial Conditions (`src/initialConditions/`):** This directory includes classes and functions for setting up the initial conditions of the simulation. This modular approach allows for easy modifications and additions of new initial condition scenarios. Initial condition functions and source functions are managed using `muparser`, which enables users to define and parse custom mathematical expressions for these parameters.

- **Matrix Operations (`src/matrix/`):** Leveraging the `Eigen` library, this module handles matrix operations, including the management of sparse matrices which are critical for efficient numerical computations in large-scale simulations.

- **Solvers (`src/solvers/`):** This directory contains the various numerical solvers used in the project. A base solver class provides a common interface, while specific solvers like the `Carleman`

`Solver`, `Euler Solver`, `ODE45`, and `PDE Solver` inherit from this base class, ensuring a consistent structure across different solving techniques.

- **Error Analysis (`src/errorAnalysis/`):** This component includes tools for analyzing the accuracy and stability of the numerical solutions. This is crucial for validating the results and refining the simulation methods.
- **Plotting and Visualization (`src/plots/`):** The `Boost` library and `gnuplot` are utilized here for generating and saving plots. This module allows for the visualization of results, making it easier to interpret and present the outcomes of the simulations.
- **Utilities (`src/utils/`):** This directory contains miscellaneous utility functions that support the overall functionality of the project, including file management and helper functions.

3.2 External Libraries and Their Roles

The project leverages several external libraries to enhance functionality and streamline the development process:

- **Eigen:** Used extensively for matrix operations, particularly for handling sparse matrices, which are crucial for the efficient computation of large systems.
- **Boost:** Employed during the plotting phase to enhance the capability of data handling and interfacing with plotting tools.
- **nlohmann/json:** Facilitates the management of JSON files, allowing for flexible and human-readable configuration options.
- **GetPot:** Used for reading and parsing `.pot` configuration files, providing a straightforward way to manage simulation parameters.
- **muparser:** A fast mathematical expression parser library, used for reading and managing functions passed as parameters in simulations, enabling users to define custom mathematical expressions.
- **gnuplot:** Integrated for saving and generating plots, aiding in the visualization of simulation results.

The combination of these components and libraries ensures that the project is robust, modular, and capable of handling complex simulations efficiently. The following sections will provide detailed explanations of the most critical classes and code segments that implement this design.

3.3 Code implementation

The `main.cpp` file serves as the entry point for the library, handling command-line arguments, reading simulation parameters from a file, and initializing and executing the main simulation. It provides options for displaying help, enabling verbose output, and specifying a parameter file.

main.cpp

```cpp
#include "GetPot" // for reading parameters
#include "MainSimulation.hpp"
#include "params/SimulationParameters.hpp"
#include "params/readParameters.hpp"
#include <iostream> // input output

int main(int argc, char **argv)
{
  GetPot cl(argc, argv);
```

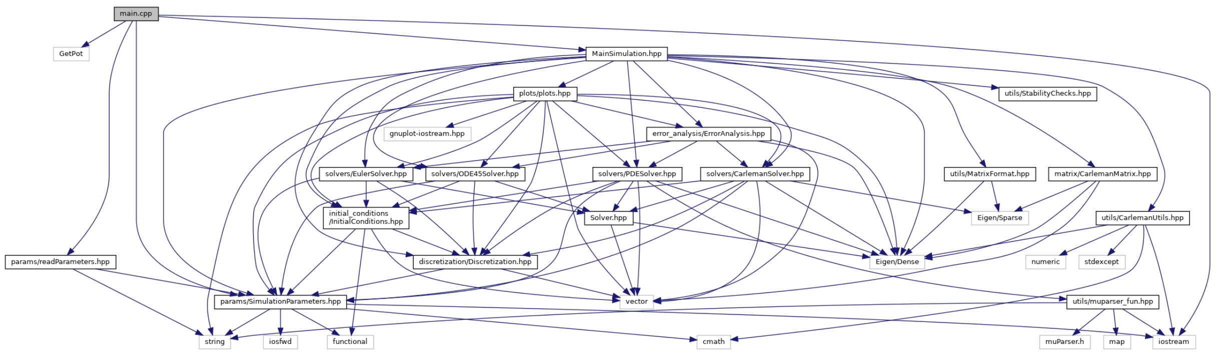

FIGURE 1
Dependency graph for `main.cpp`

```
  if(cl.search(2, "-h", "--help"))
    {
      printHelp();
      return 0;
    }
  // check if we want verbosity
  bool verbose = cl.search(1, "-v");
  // Get file with parameter values
  std::string filename = cl.follow("data/parameters.pot", "-p");

  sim::params::SimulationParameters param;
  param = sim::params::readParameters(filename, verbose);

  // Pass param as a reference to MainSimulation
  sim::MainSimulation simulation(param);

  simulation.initialize();
  simulation.run();
  return 0;
}
```

3.3.1 Main Simulation

The `simulation.cpp` file handles the main simulation logic. It constructs and initializes key components such as solvers and discretization, evaluates stability conditions, runs the Carleman and PDE solvers, and performs error analysis. This file is central to executing the numerical simulations in the library.

Initialization of solvers and components

```
// Constructs the MainSimulation object
// and initializes solvers and components.
MainSimulation::MainSimulation(params::SimulationParameters &params)
  : params(params), discretization(params),
    initialConditions(params, discretization),
    eulerSolver(params, discretization, initialConditions),
    ode45Solver(params, discretization, initialConditions),
    pdeSolver(params, discretization, initialConditions),
    carlemanSolver(params, discretization, initialConditions),
    errorAnalysis(eulerSolver, ode45Solver, pdeSolver, carlemanSolver, params)
{}
```

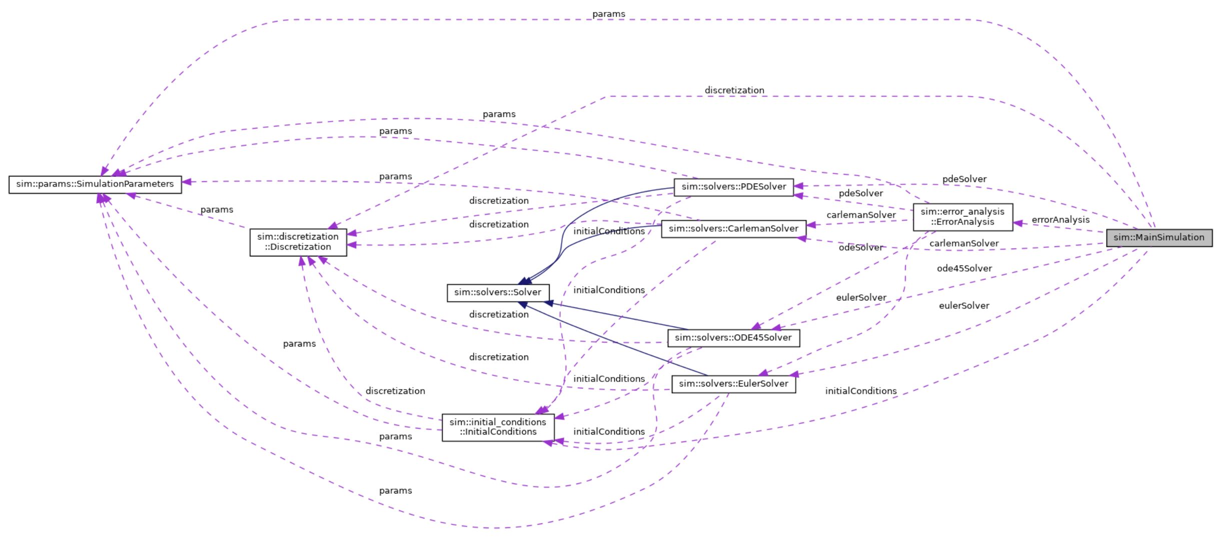
sim::params::SimulationParameters
sim::discretization ::Discretization
sim::solvers::Solver
sim::solvers::PDESolver
sim::solvers::CarlemanSolver
sim::solvers::ODE45Solver
sim::solvers::EulerSolver
sim::initial_conditions ::InitialConditions
sim::error_analysis ::ErrorAnalysis
sim::MainSimulation
params
discretization
initialConditions
pdeSolver
carlemanSolver
odeSolver
ode45Solver
eulerSolver
errorAnalysis

FIGURE 2
Collaboration diagram for `sim::MainSimulation`

Simulation Initialization

```cpp
// Sets up discretization and initial conditions.
void MainSimulation::initialize()
{
  params.initialize();
  discretization.createDiscretization();
  checkStabilityConditions();
  initialConditions.computeInitialConditions();
  initialConditions.computeForcingBoundaryConditions();
  F0 = matrixUtils::convertToDenseEigen(initialConditions.getF0());
  F1 = matrixUtils::convertToDenseEigen(initialConditions.getF1());
  F2 = matrixUtils::convertToDenseEigen(initialConditions.getF2());
}
```

Running the Simulation

```cpp
// Runs the simulation and solves the systems using different methods.
void MainSimulation::run()
{
  carlemanSolver.solve(F0, F1, F2);
  eulerSolver.solve(F0, F1, F2);
  ode45Solver.solve(F0, F1, F2);
  pdeSolver.solve(F0, F1, F2);
  errorAnalysis.computeErrors();
}
```

Stability Condition Check

```cpp
// Checks stability conditions, such as CFL conditions.
void MainSimulation::checkStabilityConditions()
{
  try
  {
    sim::stability::checkCFLConditions(
      params.U0,
      discretization.getTs()[1] - discretization.getTs()[0],
      discretization.getXs()[1] - discretization.getXs()[0],
      params.nu,
      (params.T) / (params.nt * 10 - 1),
      (params.L0) / (params.nx_pde - 1),
      (params.T) / (params.nt_pde - 1)
    );
  }
  catch(const std::runtime_error &e)
  {
    throw; // Terminate the simulation if conditions not met
  }
}
```

Carleman Number Evaluation

```
// Evaluates the Carleman convergence number.
void MainSimulation::evaluateCarlemanNumber()
{
  double R = sim::utils::calculateCarlemanConvergenceNumber(
    F0, F1, F2, initialConditions.getU0s(),
    discretization.getTs()[1] - discretization.getTs()[0],
    params.nt, params.nx, params.N_max);
  std::cout << "Carleman convergence number R: " << R << std::endl;
}
```

3.3.2 PARAMS

The SimulationParameters.hpp and SimulationParameters.cpp files define a struct, SimulationParameters, which holds the key parameters used throughout the simulation. These include spatial and temporal discretization for Euler and PDE solvers, as well as various physical constants such as Reynolds number, domain length, and viscosity. Additionally, it includes strings representing mathematical expressions for initial conditions and source functions.

The readParameters.cpp file contains functions that read and process simulation parameters from different file formats, specifically JSON and GetPot files. Based on the file extension, the appropriate function is called to extract parameters and apply default values if necessary. If the file is not found or unreadable, default values are used and appropriate messages are displayed.

Simulation Parameters Struct

```
// Structure holding the key parameters for the simulation.
struct SimulationParameters
{
  int nx = 16;          // Spatial discretization for Euler's method
  int nt = 4000;        // Temporal discretization for Euler's method
  ...

  void initialize();
};
```

Determine File Type

```
// Detects the file type (JSON or GetPot) based on the extension
// and calls the respective function
```

```cpp
SimulationParameters readParameters(const std::string &filename,
                                    bool verbose)
{
  ...
  sim::params::SimulationParameters param;
  if(jsonfile)
    param = sim::params::readParameters_json(filename, verbose);
  else
    param = sim::params::readParameters_pot(filename, verbose);
  return param;
}
```

3.3.3 Discretization

The `Discretization.cpp` file defines the `Discretization` class, which is responsible for setting up the spatial and temporal grids required for numerical simulation. It calculates the discretization for different methods such as Euler and PDE solvers, as well as for ODEs. The discretization includes grids for space and time for both PDE and ODE systems, ensuring accurate numerical solutions.

Create Discretization

```cpp
// Function to create the spatial and temporal discretization grids.
void Discretization::createDiscretization()
{
  const auto &[nx, nt, nx_pde, nt_pde, Re0, L0, U0, nu,
  beta, f, T, N_max, ode_deg, U0_fun, F0_fun] = getParams();

  double x0 = -L0 / 2;
  double x1 = L0 / 2;
  double t0 = 0;
  double t1 = T;

  // Discretization for Carleman
  dx = (x1 - x0) / (nx - 1);
  dt = (t1 - t0) / (nt - 1);

  for(int i = 0; i < nx; ++i)
    xs[i] = x0 + i * dx;
  for(int i = 0; i < nt; ++i)
    ts[i] = t0 + i * dt;

  // Discretization for PDE
  ...

  // Discretization for ODE
  int nt_ode = nt * 10; // More accurate than the PDE solution
  ...
}
```

3.3.4 Initial Conditions

The `InitialConditions.cpp` file defines the initialization and computation of initial conditions for the simulation. It leverages symbolic expressions, provided as strings, to calculate initial conditions for velocity fields (`U0`) and forcing terms (`F0`). The file also sets up the matrices `F1` and `F2`, which are used in the numerical methods for solving the system. The file uses the `MuParser` utility for evaluating mathematical expressions dynamically based on parameters.

Initial Conditions Constructor

```cpp
// Constructor initializing the initial conditions and symbolic expressions
InitialConditions::InitialConditions(
  const sim::params::SimulationParameters   &params,
  const sim::discretization::Discretization &discretization)
  : params(params), discretization(discretization)
{
  F0_expr = params.F0_fun;
  U0_expr = params.U0_fun;
}
```

Computing Initial Conditions

```cpp
// Computes initial velocity field u0 based on the given expression.
void InitialConditions::computeInitialConditions()
{
  ...
  std::map<std::string, double *> u0_vars = {
    {"x", &x_var}, {"U0", &U0}, {"L0", &L0}, {"f", &f}};
  utils::MuparserFun u0_fun(U0_expr, u0_vars);

  std::transform(xs.begin(), xs.end(), u0s.begin(), [&](double x) {
    std::map<std::string, double> u0_values = {{"x", x}};
    return u0_fun(u0_values);
  });
}
```

Forcing Boundary Conditions

```cpp
// Computes the forcing boundary conditions for F0.
void InitialConditions::computeForcingBoundaryConditions()
{
  ...
  std::map<std::string, double *> F0_vars = {
    {"t", &t_var}, {"x", &x_var}, {"U0", &U0}, {"L0", &L0}};
  utils::MuparserFun F0_fun(F0_expr, F0_vars);

  for(int it = 0; it < nt; ++it)
  {
    std::transform(xs.begin(), xs.end(), F0[it].begin(), [&](double x) {
      std::map<std::string, double> F0_values = {{"t", ts[it]}, {"x", x}};
      return F0_fun(F0_values);
    });
  }
}
```

Computation of F1 and F2

```cpp
// Computes the F1 matrix (used in discretized PDE system)
void InitialConditions::computeF1()
{
  ...
  F1.resize(nx, std::vector<double>(nx, 0));
  for (int i = 1; i < nx - 1; ++i) {
    F1[i][i] = -2 * nu / (dx * dx) - params.beta;
    F1[i][i - 1] = nu / (dx * dx);
    F1[i][i + 1] = nu / (dx * dx);
  }
  // Dirichlet boundaries
  F1[0] = std::vector<double>(nx, 0);
  F1[nx - 1] = std::vector<double>(nx, 0);
}

// Computes the F2 matrix (higher-order terms in the system)
void InitialConditions::computeF2()
{
  ...
  F2.resize(nx, std::vector<double>(nx * nx, 0));
  for (int i = 0; i < nx; ++i) {
    size_t index1 = (i - 1) * nx + i - 1;
    size_t index2 = (i + 1) * nx + i + 1;
    if ((index1 < nx * nx) && (index2 < nx * nx)) {
      F2[i][index1] = 1 / (4 * dx);
      F2[i][index2] = -1 / (4 * dx);
    }
  }
  // Dirichlet boundaries
  std::fill(F2[0].begin(), F2[0].end(), 0);
  std::fill(F2[nx - 1].begin(), F2[nx - 1].end(), 0);
}
```

3.3.5 Matrix

The `kronp.cpp` file implements matrix operations such as the Kronecker product and Kronecker power for sparse matrices using the Eigen library. These operations are essential for assembling large-scale systems in numerical simulations.

Kronecker Product of Sparse Matrices

```
// Calculates the Kronecker product of two sparse matrices
Eigen::SparseMatrix<double>
kron(const Eigen::SparseMatrix<double> &A,
     const Eigen::SparseMatrix<double> &B)
{
  ...
  Eigen::SparseMatrix<double> C(aRows * bRows, aCols * bCols);

  // Iterate over elements of A to compute the Kronecker product
  for(int k = 0; k < A.outerSize(); ++k)
  {
    for(Eigen::SparseMatrix<double>::InnerIterator it(A, k); it; ++it)
    {
      matrixUtils::assignSparseBlock(C, it.value() * B, it.row() * bRows,
                                     it.col() * bCols);
    }
  }

  return C;
}
```

Kronecker Power of a Sparse Matrix

```
// Computes the Kronecker power of a matrix by applying the Kronecker
// product k times
Eigen::SparseMatrix<double> kronp(const Eigen::SparseMatrix<double> &A,
                                  int k)
{
  Eigen::SparseMatrix<double> B(1, 1);
  B.insert(0, 0) = 1.0;
  for(int i = 0; i < k; ++i)
    B = kron(B, A);

  return B;
}
```

The `CarlemanMatrix.cpp` file uses these Kronecker operations to assemble the Carleman linearization matrix, which is vital for simulating nonlinear systems through truncation methods.

Calculate Block Sizes

```
// Calculates the block sizes required for assembling the Carleman matrix
std::vector<int> calculateBlockSizes(int N_max, int nx)
{
  std::vector<int> dNs(N_max);
  for(int N = 1; N <= N_max; ++N)
  {
    int size = static_cast<int>((std::pow(nx, N + 1) - nx) / (nx - 1));
    dNs[N - 1] = size;
  }
  return dNs;
}
```

Assemble Carleman Matrix

```
// Assembles the Carleman linearization matrix for a nonlinear system
Eigen::SparseMatrix<double>
assembleCarlemanMatrix(int N_max, int nx, int ode_deg,
                       const Eigen::MatrixXd &F0, const Eigen::MatrixXd &F1,
                       const Eigen::MatrixXd &F2)
{
  std::vector<int> dNs = calculateBlockSizes(N_max, nx);
  int total_size = dNs.back();
  Eigen::SparseMatrix<double> A(total_size, total_size);

  Eigen::MatrixXd Fs(F1.rows(), 1 + F1.cols() + F2.cols());
  Eigen::MatrixXd firstColumn = F0.row(0).transpose();
  Fs << firstColumn, F1, F2;

  Eigen::SparseMatrix<double> Fs_sparse = Fs.sparseView();
  Eigen::SparseMatrix<double> I(nx, nx);
  I.setIdentity();

  for(int i = 1; i <= N_max; ++i)
  {
    for (int j = 0; j <= std::min(ode_deg, N_max - i + 1); ++j)
```

```cpp
    {
      if (i == 1 && j == 0)
          continue;

      int a0 = 1 + (std::pow(nx, i) - nx) / (nx - 1) - 1;
      int a1 = a0 + std::pow(nx, i) - 1;
      int b0 = 1 + (std::pow(nx, j + i - 1) - nx) / (nx - 1) - 1;
      int b1 = b0 + std::pow(nx, j + i - 1) - 1;

      Eigen::SparseMatrix<double> Aij(std::pow(nx, i),
                                      std::pow(nx, j + i - 1));

      int f0 = 1 + (std::pow(nx, j) - nx) / (nx - 1) + 1;
      int f1 = f0 + std::pow(nx, j) - 1;
      Eigen::SparseMatrix<double> Fj =
        Fs_sparse.block(0, f0 - 1, Fs.rows(), f1 - f0 + 1);

      for (int p = 1; p <= i; ++p)
      {
        Eigen::SparseMatrix<double> Ia = kronp(I, p - 1);
        Eigen::SparseMatrix<double> Ib = kronp(I, i - p);
        Eigen::SparseMatrix<double> kron_product = kron(Ia, Fj);
        Aij += kron(kron_product, Ib);
      }

      matrixUtils::assignSparseBlock(A, Aij, a0, b0);
    }
  }

  return A;
}
```

3.3.6 Utils

The MuparserFun.hpp file defines the MuparserFun class, which wraps around the muParser library to simplify the parsing and evaluation of mathematical expressions. This class allows for the definition of expressions with variables and provides an easy interface to evaluate those expressions dynamically with different variable values.

MuparserFun Class Definition

```cpp
// Wrapper around the muParser library for parsing
// and evaluating expressions
class MuparserFun
{
public:
    // Copy constructor
    MuparserFun(const MuparserFun &m) : m_parser(m.m_parser), m_vars(m.m_vars)
      for (auto &var : m_vars)
          m_parser.DefineVar(var.first, var.second);

    // Constructor to initialize with an expression and variables
    MuparserFun(const std::string &expr,
                const std::map<std::string, double *> &variables)
    {
      for (const auto &var : variables)
      {
        m_parser.DefineVar(var.first, var.second);
        m_vars[var.first] = var.second;
      }
      m_parser.DefineConst("pi", 3.141592653589793);
      m_parser.SetExpr(expr);
    }

    // Evaluates the expression with given variable values
    double operator()(const std::map<std::string, double> &values)
    {
      for (const auto &val : values)
        *m_vars[val.first] = val.second;
      return m_parser.Eval();
    }

private:
    std::map<std::string, double *> m_vars; // Map of variable names to
                                            // their corresponding pointers
    mu::Parser m_parser; // muParser object for parsing
                         // and evaluating expressions
};
```

The `StabilityChecks.cpp` file contains functions that check the stability conditions for numerical simulations, particularly the Courant-Friedrichs-Lewy (CFL) conditions. It ensures that the time step and spatial discretization in the simulation meet stability limits to avoid numerical instabilities.

Check CFL Conditions

```
// Function to check CFL conditions for stability
void checkCFLConditions(double U0, double dt, double dx, double nu,
                        double dt_ode, double dx_pde, double dt_pde)
{
  double C1_e = U0 * dt / dx;
  double C2_e = 2 * nu * dt / (dx * dx);
  double C1_ode, C2_ode, C1_pde, C2_pde = ...

  // Check Euler method stability conditions
  if (C1_e > 1)
    error_message << "C1_e = " << C1_e << " exceeds stability limit.\n";
  if (C2_e > 1)
    error_message << "C2_e = " << C2_e << " exceeds stability limit.\n";

  // Check ODE solver and PDE solver stability conditions
  ...
}
```

The `MatrixFormat.cpp` file provides utility functions for matrix manipulations in the simulation. It includes functions for converting 2D vectors to Eigen dense matrices and for assigning blocks of data to sparse matrices. These functions are essential for handling large matrices in an efficient way within the simulation framework.

Assign Block to Sparse Matrix

```
// Assigns a sparse matrix block to a larger sparse matrix
// at the given position
```

```cpp
void assignSparseBlock(Eigen::SparseMatrix<double> &A,
                       const Eigen::SparseMatrix<double> &Aij,
                       int a0, int b0)
{
  A.reserve(A.nonZeros() + Aij.nonZeros());

  for (int i = 0; i < Aij.outerSize(); ++i)
  {
    for (Eigen::SparseMatrix<double>::InnerIterator it(Aij, i); it; ++it)
    {
      int row = a0 + it.row();
      int col = b0 + it.col();
      double value = it.value();

      A.coeffRef(row, col) = value;
    }
  }

  A.makeCompressed(); // Compress the matrix after the block assignment
}
```

The `CarlemanUtils.cpp` file provides utilities for analyzing matrices, particularly in the context of Carleman linearization. It includes functions to compute eigenvalues, spectral norms, and the Carleman convergence number, which is essential for assessing the stability and convergence of numerical simulations using Carleman truncation.

Eigenvalue Calculation

```cpp
// Computes the eigenvalues of a given matrix
Eigen::VectorXcd eig(const Eigen::MatrixXd &matrix)
{
  Eigen::EigenSolver<Eigen::MatrixXd> solver(matrix);
  Eigen::VectorXcd eigenvalues = solver.eigenvalues();
  return eigenvalues;
}
```

Spectral Norm Calculation

```cpp
// Computes the spectral norm (largest singular value) of a matrix
double spectralNorm(const Eigen::MatrixXd &matrix)
{
  Eigen::JacobiSVD<Eigen::MatrixXd> svd(matrix);
  return svd.singularValues()(0); // The largest singular value
}
```

Calculate Carleman Convergence Number

```cpp
// Computes the Carleman number for assessing numerical stability
double calculateCarlemanConvergenceNumber(const Eigen::MatrixXd &F0,
                                          const Eigen::MatrixXd &F1,
                                          const Eigen::MatrixXd &F2,
                                          const std::vector<double> &u0s,
                                          double dt, int not,
                                          int nx, int N_max)
{
  // Calculate eigenvalues of F1
  Eigen::VectorXcd lambdas = eig(F1);

  // Remove near-zero eigenvalues
  Eigen::VectorXcd nonZeroLambdas;
  for (int i = 0; i < lambdas.size(); ++i)
  {
    if (std::abs(lambdas[i]) > 1e-12)
    {
      nonZeroLambdas.conservativeResize(nonZeroLambdas.size() + 1);
      nonZeroLambdas(nonZeroLambdas.size() - 1) = lambdas[i];
    }
  }

  // Find the maximum eigenvalue
  double lambda = nonZeroLambdas.real().maxCoeff();

  // Calculate the norms of F0, F1, and F2
  double f2 = spectralNorm(F2);
  double f1 = spectralNorm(F1);
  double f0 = 0.0;
  for (int it = 0; it < nt; ++it)
  {
    double currentNorm = F0.row(it).norm();
    if (currentNorm > f0)
      f0 = currentNorm;
  }

  // Calculate Carleman convergence number R
  double u0sNorm = Eigen::Map<const Eigen::VectorXd>(u0s.data(),
                                                      u0s.size()).norm();
  double R = (u0sNorm * f2 + f0 / u0sNorm) / std::abs(lambda);
  ...
  return R;
}
```

3.3.7 Solvers

The `Solver.hpp` and `Solver.cpp` files define the base class for solvers in the simulation framework. The `Solver` class provides an interface for different solvers by defining a pure virtual `solve` function, which must be implemented by any derived class. Additionally, it provides a utility function, `interp1`, for performing linear interpolation at a given time point. This base class facilitates code reuse across various solvers in the system.

Solver Class Declaration

```cpp
class Solver
{
public:
  // Virtual destructor
  virtual ~Solver() = default;
  virtual void solve(Eigen::MatrixXd &F0, Eigen::MatrixXd &F1,
                     Eigen::MatrixXd &F2) = 0;
  Eigen::VectorXd interp1(const std::vector<double> &ts,
                          const Eigen::MatrixXd &F0, double t) const;
protected:
```

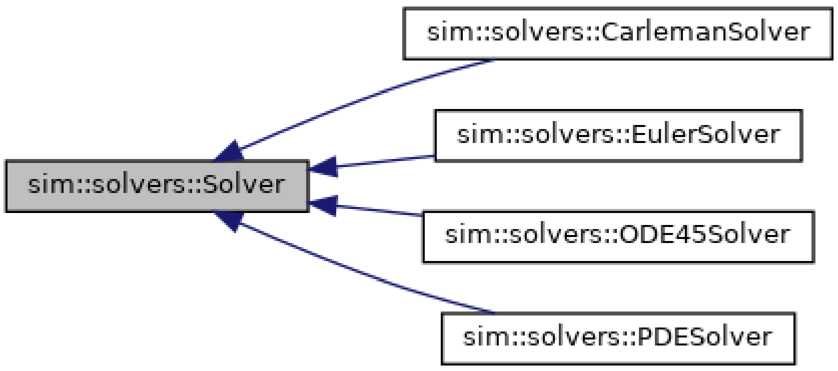
sim::solvers::CarlemanSolver
sim::solvers::EulerSolver
sim::solvers::Solver
sim::solvers::ODE45Solver
sim::solvers::PDESolver

FIGURE 3
Inheritance Diagram for `sim::solvers::Solver`

```
  Solver() = default;
};
```

1D Interpolation Function

```
// Performs linear interpolation between time steps in a matrix
// of function values
Eigen::VectorXd Solver::interp1(const std::vector<double> &ts,
                                const Eigen::MatrixXd &F0,
                                double t) const
{
  int n = ts.size();
  if (t < ts.front() || t > ts.back())
    throw std::out_of_range("t is out of bounds");

  int i = 0;
  while (i < n - 1 && t >= ts[i + 1]) ++i;

  if (t == ts[i])
    return F0.row(i);
  else if (t == ts[i + 1])
    return F0.row(i + 1);

  double t1 = ts[i];
  double t2 = ts[i + 1];
  double ratio = (t - t1) / (t2 - t1);

  return (1 - ratio) * F0.row(i) + ratio * F0.row(i + 1);
}
```

CARLEMAN SOLVER

The `CarlemanSolver.cpp` file defines the `CarlemanSolver` class, which implements a solver using Carleman linearization to approximate solutions to nonlinear systems. It constructs the Carleman matrix and solves it over time steps, storing results for each order of truncation (`N`) up to a maximum truncation level. This solver makes extensive use of sparse matrix operations and Kronecker products to handle large systems efficiently.

CarlemanSolver Constructor

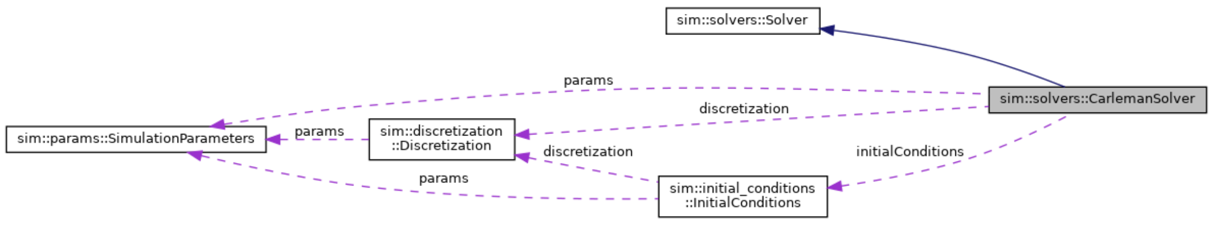
sim::solvers::Solver
params
discretization
sim::solvers::CarlemanSolver
sim::params::SimulationParameters
params
sim::discretization ::Discretization
discretization
initialConditions
params
sim::initial_conditions ::InitialConditions

FIGURE 4
Collaboration Diagram for `sim::solvers::CarlemanSolver`

```cpp
// Constructor for CarlemanSolver class
CarlemanSolver::CarlemanSolver(
  const params::SimulationParameters &params,
  const discretization::Discretization &discretization,
  const initial_conditions::InitialConditions &initialConditions)
  : params(params), discretization(discretization),
    initialConditions(initialConditions), us_c_N(params.N_max)
{}
```

Prepare Carleman Matrix

```cpp
// Prepares the Carleman linearization matrix
Eigen::SparseMatrix<double>
CarlemanSolver::prepareCarlemanMatrix(Eigen::MatrixXd &F0,
                                      Eigen::MatrixXd &F1,
                                      Eigen::MatrixXd &F2)
{
  return matrix::assembleCarlemanMatrix(params.N_max, params.nx,
                                        params.ode_deg, F0, F1, F2);
}
```

Solve Using Carleman Linearization

```cpp
// Solves the system using Carleman linearization
void CarlemanSolver::solve(Eigen::MatrixXd &F0, Eigen::MatrixXd &F1,
                           Eigen::MatrixXd &F2)
{
  Eigen::SparseMatrix<double> carleman_matrix = prepareCarlemanMatrix(F0, F1, F2);
  ...
  std::vector<int> dNs = matrix::calculateBlockSizes(N_max, nx);

  // Loop over truncation levels N and solve the Carleman system
  for (int N = 1; N <= N_max; ++N)
  {
    int dN = dNs[N - 1];
    Eigen::SparseMatrix<double> A_N = carleman_matrix.block(0, 0, dN, dN);
    Eigen::SparseMatrix<double> b_N(dN, 1);

    matrixUtils::assignSparseBlock(b_N, F0_fun(ts(0), xs).sparseView(), 0, 0);
    Eigen::MatrixXd y0s = Eigen::MatrixXd::Zero(nt, dN);

    // Solve for each time step
    std::vector<Eigen::MatrixXd> ys(nt);
    ys[0] = y0s;
    for (int k = 0; k < nt - 1; ++k)
    {
      double current_t = ts(k);
      matrixUtils::assignSparseBlock(
          b_N, F0_fun(current_t, xs).sparseView(), 0, 0);
      ys[k + 1] = ys[k] + dt * (A_N * ys[k] + b_N);
    }

    us_c_N[N - 1] = Eigen::MatrixXd(nt, nx);
    for (int k = 0; k < nt; ++k)
      us_c_N[N - 1].row(k) = ys[k].block(0, 0, nx, 1).transpose();
  }
}
```

Euler Solver

The `EulerSolver.cpp` file defines the `EulerSolver` class, which implements a solver using the explicit Euler method. The class solves the Burgers' equation or a similar nonlinear PDE using a time-stepping approach. It computes the solution matrix by iterating over each time step, applying the Euler method to approximate the solution at the next time step.

EulerSolver Constructor

```
// Constructor for EulerSolver class
EulerSolver::EulerSolver(
  const params::SimulationParameters                &params,
  const sim::discretization::Discretization         &discretization,
  const sim::initial_conditions::InitialConditions &initialConditions)
  : params(params), discretization(discretization),
    initialConditions(initialConditions),
    us_e(Eigen::MatrixXd::Zero(params.nt, params.nx))
{}
```

Solve Using Euler's Method

```
// Solves the system using the explicit Euler method
void EulerSolver::solve(Eigen::MatrixXd &F0, Eigen::MatrixXd &F1,
                        Eigen::MatrixXd &F2)
{
  double dt = discretization.getTs()[1] - discretization.getTs()[0];

  auto F0_interp = [&](double t) {
    Eigen::VectorXd interp = interp1(discretization.getTs(), F0, t);
    return interp;
  };

  auto burgers_odefun = [&](double t, const Eigen::MatrixXd &u) {
    Eigen::VectorXd burger_fun =
      F0_interp(t) + F1 * u + F2 * matrix::kron(u.sparseView(),
                                                  u.sparseView());
    return burger_fun;
  };

  std::cout << "Solving direct Euler" << std::endl;

  Eigen::MatrixXd u0s = Eigen::Map<const Eigen::MatrixXd>(
    initialConditions.getU0s().data(), initialConditions.getU0s().size(), 1);

  us_e.row(0) = u0s.transpose();
  std::vector<double> ts = discretization.getTs();

  for (int k = 0; k < params.nt - 1; ++k)
  {
    us_e.row(k + 1) =
      us_e.row(k) +
      dt * burgers_odefun(ts[k], us_e.block(k, 0, 1, params.nx).transpose())
             .transpose()
             .row(0);
  }
}
```

ODE45 Solver

The `ODE45Solver.cpp` file defines the `ODE45Solver` class, which implements a solver using the classical Runge-Kutta method (RK45) to solve a system of ordinary differential equations (ODEs). It uses adaptive time-stepping to achieve accurate solutions for nonlinear systems and computes the solution matrix over a time grid.

ODE45Solver Constructor

```
// Constructor for ODE45Solver class
ODE45Solver::ODE45Solver(
  const params::SimulationParameters                &params,
  const sim::discretization::Discretization         &discretization,
  const sim::initial_conditions::InitialConditions &initialConditions)
  : params(params), discretization(discretization),
    initialConditions(initialConditions),
    us_d(Eigen::MatrixXd::Zero(params.nt, params.nx))
{}
```

Solve Using Runge-Kutta 4-5

```
// Solves the system using the classical Runge-Kutta method (RK45)
void ODE45Solver::solve(Eigen::MatrixXd &F0, Eigen::MatrixXd &F1,
                        Eigen::MatrixXd &F2)
{
  Eigen::MatrixXd us_ode = Eigen::MatrixXd::Zero(10 * params.nt, params.nx);
  Eigen::MatrixXd u0s = Eigen::Map<const Eigen::MatrixXd>(
    initialConditions.getU0s().data(), initialConditions.getU0s().size(), 1);

  us_ode.row(0) = u0s.transpose();
  std::vector<double> ts_ode = discretization.getTsOde();
  double              t = ts_ode[0];

  for (int i = 1; i < ts_ode.size(); ++i)
  {
    double h = ts_ode[i] - ts_ode[i - 1];
    Eigen::MatrixXd u = us_ode.row(i - 1).transpose();

    // Runge-Kutta steps
    Eigen::MatrixXd k1 = burgers_odefun(t, u);
    Eigen::MatrixXd k2 = burgers_odefun(t + 0.5 * h, u + 0.5 * h * k1);
    Eigen::MatrixXd k3 = burgers_odefun(t + 0.5 * h, u + 0.5 * h * k2);
    Eigen::MatrixXd k4 = burgers_odefun(t + h, u + h * k3);

    // Compute the next step
    Eigen::MatrixXd u_next = u + (h / 6.0) * (k1 + 2 * k2 + 2 * k3 + k4);
    us_ode.row(i) = u_next.transpose();
    t = ts_ode[i];
  }

  std::vector<double> ts = discretization.getTs();

  for (int i = 0; i < params.nt; ++i)
  {
    Eigen::VectorXd u_interpolated = interp1(ts_ode, us_ode, ts[i]);
    us_d.row(i) = u_interpolated;
  }
}
```

PDE Solver

The `PDESolver.cpp` file defines the `PDESolver` class, which solves partial differential equations (PDEs) using an explicit finite difference method. The class solves nonlinear PDEs by applying boundary conditions, time-stepping, and interpolation to match both the time and space grids of the simulation. The solver is responsible for solving the PDE on a coarse grid and interpolating the results to a finer grid for analysis.

PDESolver Constructor

```
// Constructor for PDESolver class
PDESolver::PDESolver(
  const params::SimulationParameters               &params,
  const sim::discretization::Discretization        &discretization,
  const sim::initial_conditions::InitialConditions &initialConditions)
  : params(params), discretization(discretization),
    initialConditions(initialConditions),
    us_pde(Eigen::MatrixXd::Zero(params.nt, params.nx))
{}
```

PDE Function Definition

```
// Defines the PDE to solve using explicit finite difference methods
double PDESolver::pde(double x, double t, double u, double dudx) const
{
  double nu = params.nu;
  double beta = params.beta;
  return nu * dudx - pow(u, 2) / 2.0 + F0_fun(t, x);
}
```

Apply Boundary Conditions

```
// Apply Dirichlet boundary conditions to the PDE solution
void
PDESolver::apply_boundary_conditions(Eigen::VectorXd &u, double t) const
{
  u(0) = 0;            // Left boundary
  u(u.size() - 1) = 0; // Right boundary
}
```

Solve PDE with Time-Stepping

```
// Solves the PDE using an explicit finite difference method
// with time-stepping
void PDESolver::solvePDE(Eigen::MatrixXd &us_pde_full)
{
  ..
  // Initialize with initial condition
  for (int i = 0; i < nx_pde; ++i)
    us_pde_full(0, i) = U0_fun(xs_pde[i]);

  // Time-stepping loop
  for (int n = 0; n < nt_pde - 1; ++n)
  {
    Eigen::VectorXd u = us_pde_full.row(n);
    Eigen::VectorXd dudx(nx_pde);

    // Central difference to compute spatial derivatives
    for (int i = 1; i < nx_pde - 1; ++i)
      dudx(i) = (u(i + 1) - u(i - 1)) / (2 * dx);

    // Apply boundary conditions
    apply_boundary_conditions(u, ts_pde[n]);

    // Update solution using explicit finite difference
    for (int i = 1; i < nx_pde - 1; ++i)
    {
      double convection = -u(i) * (u(i + 1) - u(i - 1)) / (2 * dx);
      double diffusion = params.nu *
                         (u(i + 1) - 2 * u(i) + u(i - 1)) / (dx * dx);
      double source = F0_fun(ts_pde[n], xs_pde[i]);

      us_pde_full(n + 1, i) = u(i) + dt * (convection + diffusion + source);
    }

    // Apply boundary conditions after updating
    Eigen::VectorXd next_u = us_pde_full.row(n + 1);
    apply_boundary_conditions(next_u, ts_pde[n]);
    us_pde_full.row(n + 1) = next_u;
  }
}
```

3.3.8 Error Analysis

The `ErrorAnalysis.cpp` file implements the `ErrorAnalysis` class, which calculates the errors between various numerical solutions such as Carleman linearization, ODE solvers, PDE solvers, and Euler methods. It computes both the absolute and relative errors for each time step and stores them for later analysis. The class uses L2 and L-infinity norms to evaluate the error magnitude between different solution methods.

ErrorAnalysis Constructor

```
// Constructor for the ErrorAnalysis class
ErrorAnalysis::ErrorAnalysis(...)
  : eulerSolver(eulerSolver), odeSolver(odeSolver), pdeSolver(pdeSolver),
    carlemanSolver(carlemanSolver), params(params),
    eps_c_d_N(Eigen::MatrixXd::Zero(params.N_max, params.nt)),
    eps_rel_c_d_N(Eigen::MatrixXd::Zero(params.N_max, params.nt)),
    eps_c_pde_N(Eigen::MatrixXd::Zero(params.N_max, params.nt)),
    eps_rel_c_pde_N(Eigen::MatrixXd::Zero(params.N_max, params.nt)),
    eps_d_pde(Eigen::VectorXd::Zero(params.nt)),
    eps_rel_d_pde(Eigen::VectorXd::Zero(params.nt)),
    eps_d_e(Eigen::VectorXd::Zero(params.nt)),
    eps_rel_d_e(Eigen::VectorXd::Zero(params.nt))
{}
```

Compute Errors

```
// Computes the absolute and relative errors between different solutions
void ErrorAnalysis::computeErrors()
{
  for (int N = 0; N < params.N_max; ++N)
  {
    // Absolute and relative error between Carleman and ODE solutions
    Eigen::MatrixXd dus_c_d_N = us_c_N[N] - us_d;
    Eigen::MatrixXd dus_rel_c_d_N = dus_c_d_N.array() / us_d.array();
    dus_rel_c_d_N = dus_rel_c_d_N.unaryExpr([](double v) {
      return std::isfinite(v) ? v : 0; });

    // Absolute and relative error between Carleman and PDE solutions
    Eigen::MatrixXd dus_c_pde_N = us_c_N[N] - us_pde;
    Eigen::MatrixXd dus_rel_c_pde_N = dus_c_pde_N.array() / us_pde.array();
    dus_rel_c_pde_N = dus_rel_c_pde_N.unaryExpr([](double v) {
      return std::isfinite(v) ? v : 0; });

    // Compute norms for each time step
    for (int k = 0; k < params.nt; ++k)
    {
      // L2 norm and L-inf norm for Carleman vs ODE/PDE
      eps_c_d_N(N, k) = dus_c_d_N.row(k).norm();
      eps_rel_c_d_N(N, k) = dus_rel_c_d_N.row(k).lpNorm<Eigen::Infinity>();
      eps_c_pde_N(N, k) = dus_c_pde_N.row(k).norm();
      eps_rel_c_pde_N(N, k) = dus_rel_c_pde_N.row(k).lpNorm<Eigen::Infinity>();
    }
  }
  ...
}
```

3.3.9 Plots

The `plots.cpp` file defines the `Plotter` class, which is responsible for generating visualizations of the solutions and error analysis from the simulation. It uses `gnuplot` to create plots such as solution profiles, error analysis, and convergence analysis. The class creates custom folder structures to store outputs based on the simulation parameters.

Plotter Constructor

```cpp
// Constructor for Plotter class
Plotter::Plotter(...)
    : params(params), discretization(discretization),
      initialConditions(initialConditions), carlemanSolver(carlemanSolver),
      eulerSolver(eulerSolver), pdeSolver(pdeSolver),
      ode45Solver(ode45Solver), errorAnalysis(errorAnalysis)
{}
```

Initialize Plotting Environment

```cpp
// Initialize the plotting environment and create output folders
void Plotter::initialize()
{
  std::string folder_name = "output/nx_" + std::to_string(params.nx) +
                            "_nt_" + std::to_string(params.nt) + "_Nmax_" +
                            std::to_string(params.N_max);
  boost::filesystem::path output_dir(folder_name);
  if (!boost::filesystem::exists(output_dir))
  {
    boost::filesystem::create_directories(output_dir);
  }

  gp << "set terminal pngcairo enhanced\n"; // Use default plot size
}
```

Plot Solution

```cpp
// Plot the simulation solutions
void Plotter::plotSolution()
{
  ...
  // Plot the initial condition
  std::vector<std::pair<double, double>> init_condition(nx);
  for (size_t i = 0; i < nx; ++i)
    init_condition[i] = std::make_pair(xs[i], initialConditions.getU0s()[i]);
  gp.send1d(init_condition);

  // Additional plots for source shape, Euler solution,
  // and Carleman solutions
  ...

  gp << "unset output\n";
}
```

Plot Errors

```cpp
// Plot the absolute L2 error between Carleman and ODE45 solutions
void Plotter::plotErrors()
{
  ...
  const auto &eps_c_d_N = errorAnalysis.getEpsCDError();

  for (size_t N = 0; N < eps_c_d_N.rows(); ++N)
  {
    std::vector<std::pair<double, double>> error_data;
    for (size_t k = 0; k < eps_c_d_N.cols(); ++k)
```

```cpp
      error_data.push_back(std::make_pair(discretization.getTs()[k],
                                           eps_c_d_N(N, k)));
    gp.send1d(error_data);
  }

  gp << "unset output\n";
}
```

Plot Error Convergence

```cpp
// Plot the error convergence for Carleman solutions
void Plotter::plotErrorConvergence()
{
  ...
  const auto &eps_c_d_N = errorAnalysis.getEpsCDError();
  Eigen::VectorXd max_values = eps_c_d_N.rowwise().maxCoeff();

  std::vector<std::pair<int, double>> max_error_data;
  for (size_t N = 0; N < max_values.size(); ++N)
    max_error_data.push_back(std::make_pair(N + 1, max_values[N]));

  gp.send1d(max_error_data);
  gp << "unset output\n";
}
```

4 Results

In this section, we present the results obtained from three different experiments conducted using the Carleman solver. The parameters used for the simulations are as follows:

- Spatial discretization for Euler's method: $nx = \_$
- Temporal discretization for Euler's method: $nt = \_$
- Spatial discretization for the pdepe solver: $nx\_pde = 100$
- Temporal discretization for the pdepe solver: $nt\_pde = 40000$
- Desired Reynolds number: $Re_0 = 20$
- Domain length: $L_0 = 1$
- Linear damping coefficient: $\beta = 0$
- Number of oscillations of the sinusoidal initial condition inside the domain: $f = 1$
- Simulation time: $T = 3$
- Maximum Carleman truncation level: $N_{\text{max}} = \_$
- Degree of the Carleman ODE: $ode\_deg = 2$
- Initial condition: $U_0(x) = -U_0 \sin\left(\frac{2\pi f x}{L_0}\right)$
- Source function: $F_0(x,t) = U_0 \exp\left(-\frac{(x-L_0/4)^2}{2(L_0/32)^2}\right) \cos(2\pi t)$

For all the experiments, the values of U_0 and ν are determined as:

$$U_0 = \frac{1}{nx - 1}, \quad \nu = \frac{U_0 L_0}{Re_0}$$

The experiments differ by the values of nx, nt, and N_{max} used:

1. $nx = 5$, $nt = 20$, $N_{\text{max}} = 4$
2. $nx = 9$, $nt = 400$, $N_{\text{max}} = 4$
3. $nx = 16$, $nt = 4000$, $N_{\text{max}} = 3$

For each experiment, we compute the Carleman number, which quantifies the truncation error. The computed Carleman numbers are as follows:

- Experiment 1: Carleman number = 8.245
- Experiment 2: Carleman number = 18.521
- Experiment 3: Carleman number = 43.593

Given the nature of the problem, all Carleman numbers are greater than 1, which is expected. Furthermore, the CFL conditions are respected in all cases.

4.1 Experiment 1: $nx = 5$, $nt = 20$, $N_{\text{MAX}} = 4$

In this first experiment, we use a coarse grid with $nx = 5$ and $nt = 20$. The following results are obtained (5, 6, 7):

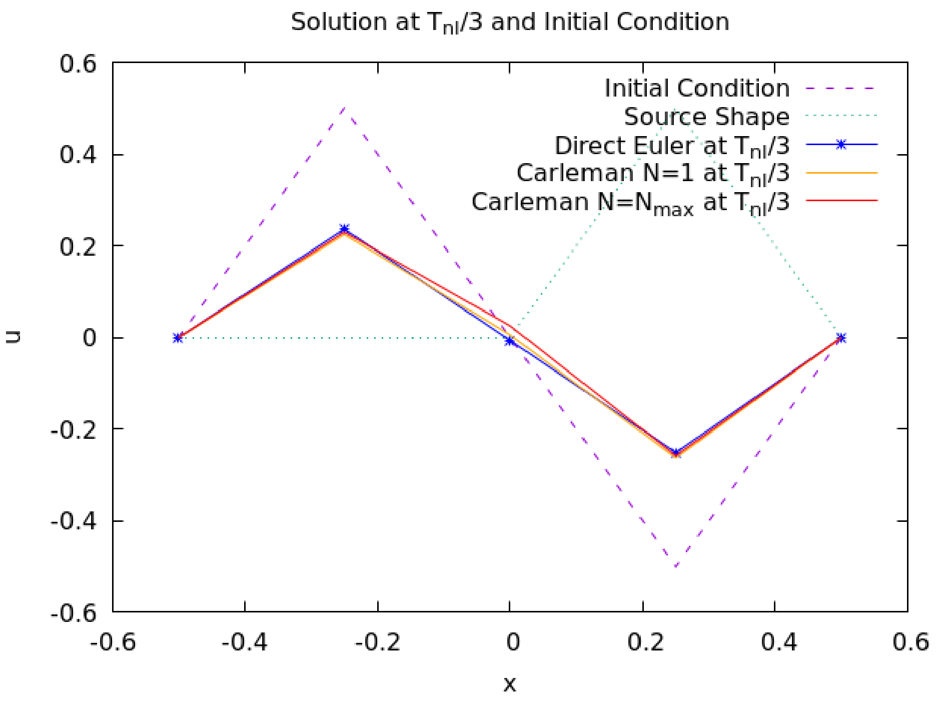

Figure 5
Solution plots for the initial condition, source function, Euler solution, Carleman solution for $N = 1$, and Carleman solution for $N = N_{\text{max}}$ at $t = T_{nl}/3$.

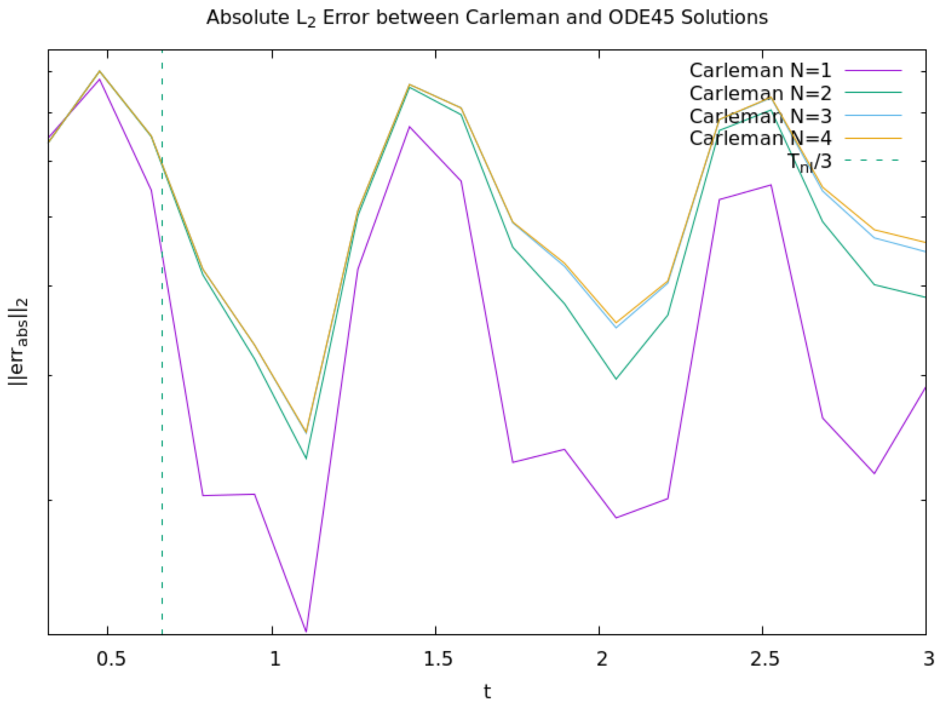

FIGURE 6
Absolute error between the Carleman solution and the ODE45 solution.

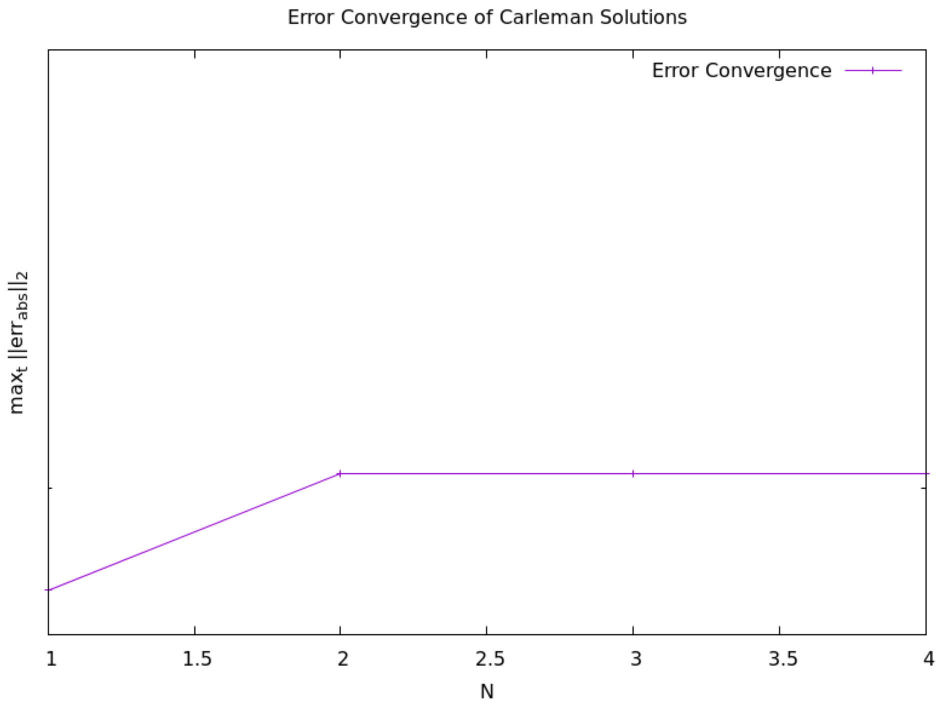

FIGURE 7
Convergence of the error with respect to the Carleman order N.

4.2 Experiment 2: $nx = 9$, $nt = 400$, $N_{\text{MAX}} = 4$

In this second experiment, we use a more refined grid with $nx = 9$ and $nt = 400$. The results obtained are as follows (8, 9, 10):

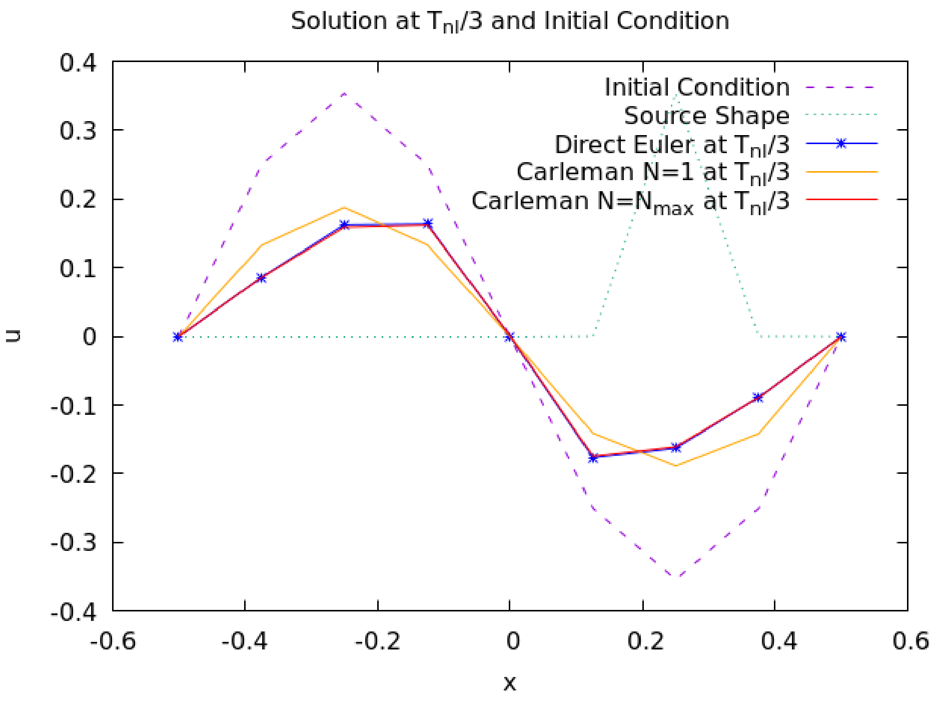

FIGURE 8
Solution plots for the initial condition, source function, Euler solution, Carleman solution for $N = 1$, and Carleman solution for $N = N_{\text{max}}$ at $t = T_{nl}/3$.

4.3 Experiment 3: $nx = 16$, $nt = 4000$, $N_{\text{MAX}} = 3$

In the final experiment, we further refine the grid with $nx = 16$ and $nt = 4000$. The results are shown below (11, 12, 13):

4.4 Discussion

The results from the three experiments highlight the effect of grid refinement and Carleman truncation on the solution accuracy. In the first simulation, with $nx = 5$ and $nt = 20$, the grid is too coarse, and we do not achieve convergence even when increasing N_{max}. This suggests that the spatial and temporal discretization is insufficient to capture the dynamics of the problem.

In contrast, for the second and third experiments, where the grid is more refined ($nx = 9$ and $nx = 16$, respectively), we observe that the error converges exponentially with respect to the Carleman order N, as expected from the theory. Even though the Carleman numbers are greater than 1, indicating a potential loss of accuracy, the convergence behavior aligns with theoretical predictions, showing exponential decay in error for higher N values.

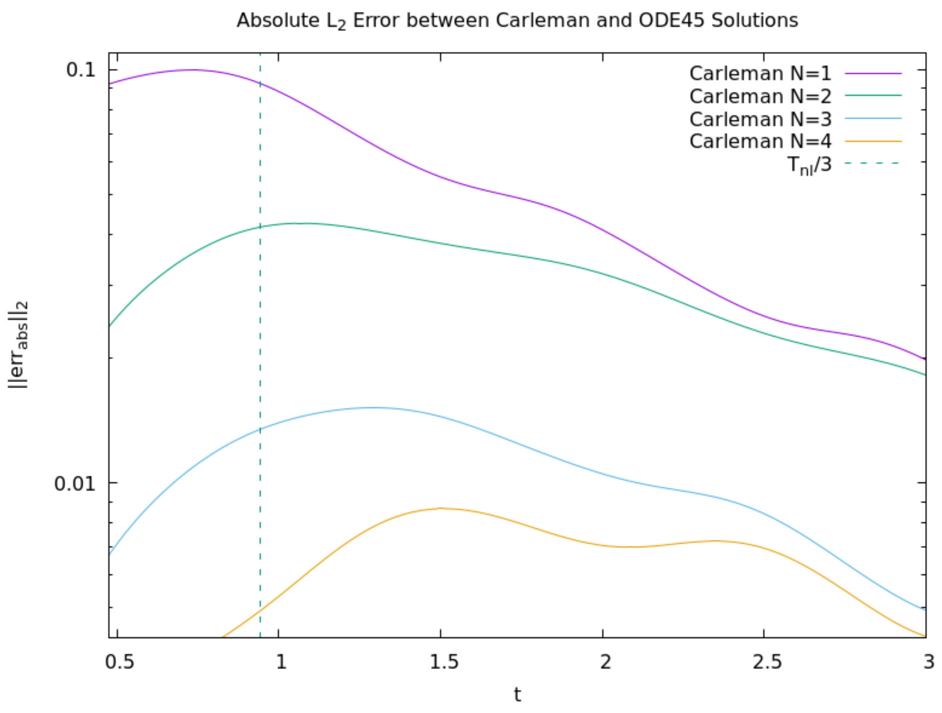

FIGURE 9
Absolute error between the Carleman solution and the ODE45 solution.

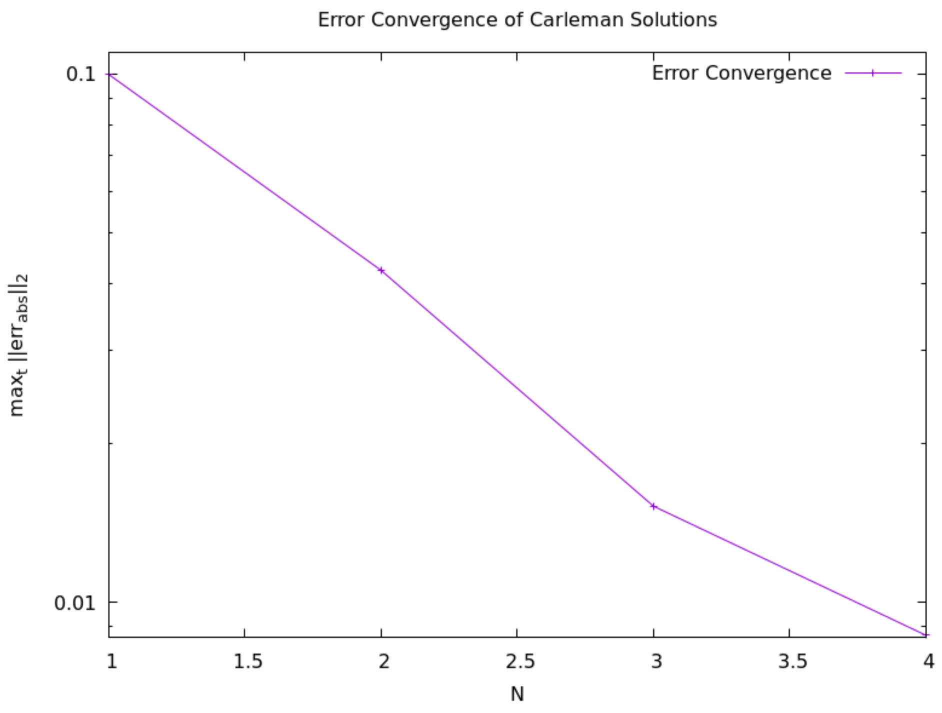

FIGURE 10
Convergence of the error with respect to the Carleman order N.

5 CONCLUSION

The main objective of this project was to create a reusable and efficient C++ library for the Carleman solver, translating the existing MATLAB code into a more advanced and performant language. The

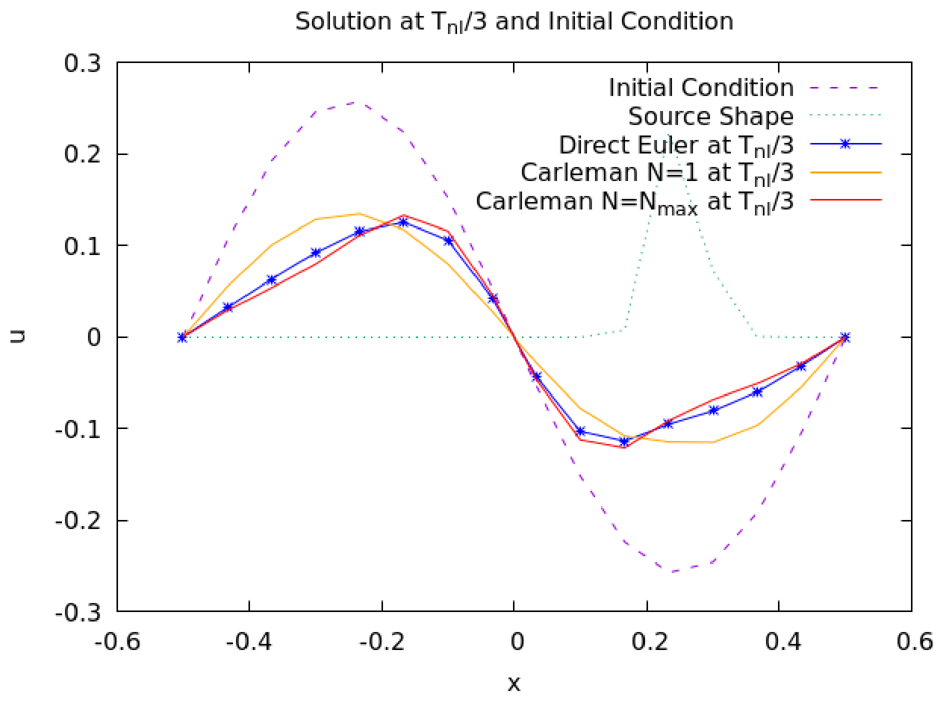

FIGURE 11
Solution plots for the initial condition, source function, Euler solution, Carleman solution for $N = 1$, and Carleman solution for $N = N_{\text{max}}$ at $t = T_{nl}/3$.

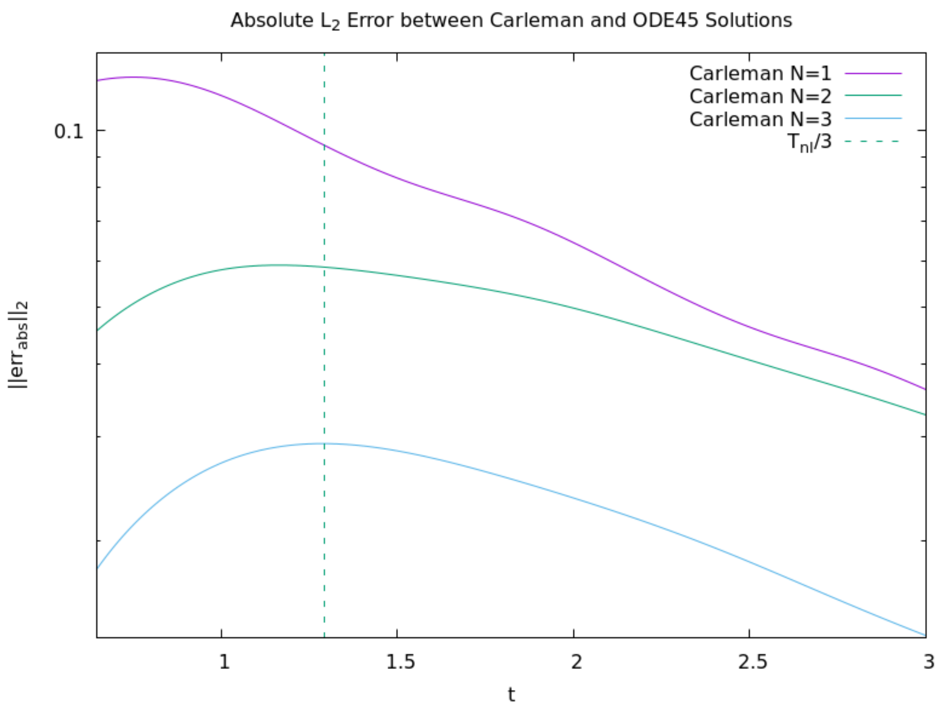

FIGURE 12
Absolute error between the Carleman solution and the ODE45 solution.

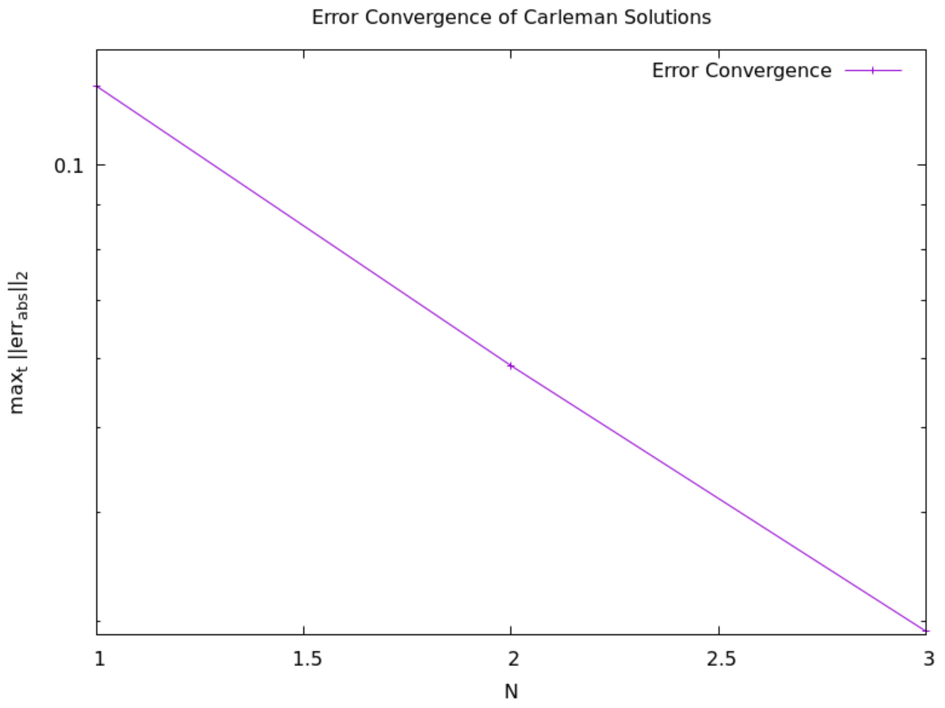

FIGURE 13
Convergence of the error with respect to the Carleman order N.

resulting C++ implementation offers improved memory management, flexibility, and computational efficiency. It is designed to be modular, allowing for easy adaptation and expansion for various problem setups. This transition opens the door for future applications of the solver to a wide range of problems in dynamical systems, particularly for solving PDEs and ODEs.

The results from the three test cases demonstrated the effectiveness of the solver while highlighting the importance of grid resolution and discretization. In the first experiment ($nx = 5$, $nt = 20$), the coarse grid led to non-convergence, even with an increased Carleman truncation level N_{max}, showing the need for finer resolution. In contrast, the second and third experiments ($nx = 9$ and $nx = 16$) exhibited exponential error convergence with N, aligning with theoretical expectations. All experiments had Carleman numbers greater than 1, yet the solver maintained reliable convergence in the second and third cases, confirming its accuracy. The CFL conditions were met, ensuring the stability of the method throughout.

While the C++ implementation of the Carleman solver is functional and performs well in many scenarios, there are several areas for improvement to enhance its performance and versatility:

- **Parallelization:** One of the most significant potential improvements would be parallelizing the computations, either through multi-threading on CPUs or GPU acceleration. This would considerably reduce the computational time for large-scale problems, particularly when using high-resolution grids or when solving problems that require a large number of time steps.
- **Integration with Numerical Libraries:** Integrating the Carleman solver with existing numerical libraries, such as PETSc, Eigen, or Boost, would enhance the efficiency of linear algebra operations and offer access to more sophisticated solvers. This could enable the implementation of implicit methods or adaptive time-stepping schemes, further improving the solver's accuracy and flexibility.
- **More Simulations and Comprehensive Analysis:** Expanding the library's test cases to include different boundary conditions, nonlinearities, and damping coefficients would provide deeper insight

into the robustness of the solver. Furthermore, a more detailed analysis of the Carleman number and its impact on the error and convergence properties could help refine the solver's tuning parameters.

In conclusion, the Carleman solver implementation in C++ achieved its primary goal of translating MATLAB code into a more advanced, reusable, and efficient library. The results showed that the solver works effectively when appropriate spatial and temporal resolutions are chosen, with exponential convergence of the error with respect to the Carleman order N. While the first test case struggled with non-convergence due to insufficient grid refinement, the subsequent cases confirmed the theoretical expectations of exponential error decay. The current library provides a strong foundation for future research, with clear avenues for further improvement, particularly through parallelization and integration with advanced numerical libraries.

References

[1] Jin-Peng Liu et al. 'Efficient quantum algorithm for dissipative nonlinear differential equations'. In: *Proceedings of the National Academy of Sciences* 118.35 (Aug. 2021). ISSN: 1091-6490. DOI: 10.1073/pnas.2026805118. URL: http://dx.doi.org/10.1073/pnas.2026805118.

[2] Pierre Gilles Lemarié-Rieusset. *The Navier-Stokes problem in the 21st century*. Chapman and Hall/CRC, 2018.

[3] Lev Kofman, Dmitri Pogosian and Sergei Shandarin. 'Structure of the universe in the two-dimensional model of adhesion'. In: *Monthly Notices of the Royal Astronomical Society* 242.2 (1990), pp. 200–208.

[4] Sergei F Shandarin and Ya B Zeldovich. 'The large-scale structure of the universe: Turbulence, intermittency, structures in a self-gravitating medium'. In: *Reviews of Modern Physics* 61.2 (1989), p. 185.

[5] Massimo Vergassola et al. 'Burgers' equation, Devil's staircases and the mass distribution for large-scale structures'. In: *Astronomy and Astrophysics (ISSN 0004-6361), vol. 289, no. 2, p. 325-356* 289 (1994), pp. 325–356.

[6] Peter Alan Davidson. *Introduction to magnetohydrodynamics*. Vol. 55. Cambridge university press, 2017.

[7] Yann Brenier and Emmanuel Grenier. 'Sticky particles and scalar conservation laws'. In: *SIAM journal on numerical analysis* 35.6 (1998), pp. 2317–2328.

[8] Min-Hang Bao. *Micro mechanical transducers: pressure sensors, accelerometers and gyroscopes*. Elsevier, 2000.

[9] Constantine M Dafermos and Constantine M Dafermos. *Hyperbolic conservation laws in continuum physics*. Vol. 3. Springer, 2005.

[10] Johannes Martinus Burgers. 'A mathematical model illustrating the theory of turbulence'. In: *Advances in applied mechanics* 1 (1948), pp. 171–199.

[11] John Charles Butcher. *Numerical methods for ordinary differential equations*. John Wiley & Sons, 2016.

[12] William F Ames. *Numerical methods for partial differential equations*. Academic press, 2014.