1D Advection Equation

Mimetic methods for solving PDEs and the MOLE library

Carlos Aznarán

March 18, 2025

One-dimensional advection partial differential equation with constant velocity

Consider the following BVP/IVP for the advection equation with a Dirichlet boundary condition

$$\text{(1)} \qquad \begin{cases} \partial_t u(x,t) + c\partial_x u(x,t) = f(x), & x \in (a,b), t > 0. \\ u(a,t) = g(t), & t > 0. \\ u(x,0) = u_0(x), & x \in (a,b). \end{cases}$$

The solution of (1) is given by

$$u(x,t) = g(x-ct) + \int_0^t f(x - c(t-\theta))\, \mathrm{d}\theta.$$

Example (Non-homogeneous)

$$\begin{cases} \partial_t u(x,t) + c\partial_x u(x,t) = 1, & x \in (-1,1), t > 0. \\ u(-1,t) = 0, & t > 0. \\ u(x,0) = x+1, & x \in (-1,1). \end{cases}$$

The solution is given by

$$u(x,t) = \begin{cases} x - ct + 1 + t, & x - ct \geq -1. \\ \frac{x+1}{c} + t - \frac{x+1}{c}, & x - ct < -1. \end{cases}$$

Example (Homogeneous)

Let's consider the program `hyperbolic1D.m` subject to the following configuration.

$$\begin{cases} \partial_t u(x,t) + a\partial_x u(x,t) = 0, & x \in (0,1), t \in (0,1). \\ u(0,t) = u(1,t) = 0, & t \in (0,1). \\ u(x,0) = \sin(2\pi x), & x \in (0,1). \end{cases}$$

The exact solution is given by

$$u(x,t) = \sin(2\pi(x - at)).$$

Leapfrog scheme

$$u_i^{n+1} = u_i^n - \frac{c\Delta t}{\Delta x}\left(u_{i+1}^n - u_{i-1}^n\right) + \mathcal{O}\left(\Delta t^2, \Delta x^2\right).$$

CFL condition

Using von Neumann stability analysis it can be shown that the Leapfrog scheme is stable when

$$\frac{|c|\,\Delta t}{\Delta x} \leq 1.$$

Program `hyperbolic1D.m` with the MOLE library

```
#!/usr/bin/env -S octave -qf
% Solves the 1D Advection equation with periodic boundary conditions

addpath('/usr/share/mole/matlab/')

a = 1; % Velocity
west = 0; % Domain's limits
east = 1;

k = 2; % Operator's order of accuracy
m = 50; % Number of cells
dx = (east - west) / m;

t = 1; % Simulation time
dt = dx / abs(a); % CFL condition for explicit schemes
alpha = abs(a) * dt / dx;

D = div(k, m, dx); % 1D Mimetic divergence operator% size(D)
save("-hdf5", "D_before.h5", "D")
figure('visible', 'off');
spy(D);
saveas(gcf, "hyperbolic1D_divergence_sparsebefore.pdf", 'pdfcrop')
I = interpol(m, 0.5); % 1D 2nd order interpolator% 0⩽c⩽1% size(I);
spy(I);
saveas(gcf, "hyperbolic1D_interpolator_sparse.pdf", 'pdfcrop')
save("-hdf5", "I_name.h5", "I")

% 1D Staggered grid
grid = [west west + dx / 2:dx:east - dx / 2 east];
size(grid);
% IC
U = sin(2 * pi * grid)'; % 100 * ones(m + 2, 1) or repmat(100, [m + 2, 1])
```

Program hyperbolic1D.m with the MOLE library

```
% Periodic BC imposed on the divergence operator
D(1, 2) = 1 / (2 * dx);
D(1, end - 1) = -1 / (2 * dx);
D(end, 2) = 1 / (2 * dx);
D(end, end - 1) = -1 / (2 * dx);

% Premultiply out of the time loop (since it doesn't change)
D = -a * dt * 2 * D * I;
% One could also have said: D = -a*dt*2*I*D if the grid
% was defined as: grid = west : dx : east (nodal)
save("-hdf5", "D_after.h5", "D")
spy(D);
saveas(gcf, "hyperbolic1D_divergence_sparseafter.pdf", 'pdfcrop')

U2 = U + D / 2 * U; % Compute one step using Euler's method

figure('visible', 'off');
% Time integration loop
for i = 1:t / dt
    plot(grid, U2, 'o-')% Plot approximation
    hold on
    plot(grid, sin(2 * pi * (grid - a * i * dt)))% Plot exact solution
    hold off
    str = sprintf('t = %.2f', i * dt);
    title(str)
    xlabel('x')
    ylabel('u(x, t)')
    axis([west east -1.5 1.5])
    pause(0.04)
    U3 = U + D * U2; % Compute next step using Leapfrog scheme
    U = U2;
    U2 = U3;
    plot_name = sprintf("hyperbolic1D%i.pdf", i);
    saveas(gcf, plot_name, 'pdfcrop')
end
```

Program hyperbolic1D_upwind.cpp with the MOLE library

```cpp
#include <cmath>
#include <iostream>
#include <mole/laplacian.h>
// #include <mole/operators.h>
// #include <mole/robinbc.h>
#include <numbers>

arma::sp_mat sidedNodalTemp(int m, double dx, const std::string &type)
{

  // Create a sparse matrix of size (m+1) x (m+1)
  arma::sp_mat S(m + 1, m + 1);
  if (type == "backward") {
    // Backward difference
    S.diag(-1) = -arma::ones<vec>(m);   // Sub-diagonal
    S.diag(0) = arma::ones<vec>(m + 1); // Main diagonal
    S(0, m - 1) = -1;                   // Wrap-around for periodic boundary
    S /= dx;
  }
  else if (type == "forward") {
    // Forward difference
    S.diag(0) = -arma::ones<vec>(m + 1); // Main diagonal
    S.diag(1) = arma::ones<vec>(m);      // Super-diagonal
    S(m, 1) = 1;                         // Wrap-around for periodic boundary
    S /= dx;
  }
  else { // "centered"
    // Centered difference
    S.diag(-1) = -arma::ones<vec>(m); // Sub-diagonal
    S.diag(1) = arma::ones<vec>(m);   // Super-diagonal
    S(0, m - 1) = -1;                 // Wrap-around for periodic boundary
    S(m, 1) = 1;                      // Wrap-around for periodic boundary
    S /= (2 * dx);
  }
```

Program hyperbolic1D_upwind.cpp with the MOLE library

```cpp
}

int main()
{
  double a = 1.0;    // Velocity
  double west = 0.0; // Domain's left limit
  double east = 1.0; // Domain's right limit

  int m = 20;                    // Number of cells
  double dx = (east - west) / m; // Grid spacing

  double t = 1.0;               // Simulation time
  double dt = dx / std::abs(a); // Time step based on CFL condition

  arma::sp_mat S = sidedNodalTemp(
      m, dx, (a > 0) ? "backward" : "forward"); // Use "forward" if a < 0

  arma::vec grid = arma::regspace(west, dx, east);
  arma::vec U = arma::sin(2 * std::numbers::pi * grid);

  S = arma::speye<arma::sp_mat>(S.n_rows, S.n_cols) - a * dt * S;

  int steps = t / dt;

  std::ofstream dataFile("results.dat");
  if (!dataFile) {
    std::cerr << "Error opening data file.\n";
    return 1;
  }

  // Write all time steps to a single data file
  for (int i = 1; i <= steps; ++i) {
    // Compute U^(n+1)
    U = S * U;
```

Program hyperbolic1D_upwind.cpp with the MOLE library

```
    // Store the data with an empty line between time steps for indexing in
    // GNUplot
    for (size_t j = 0; j < grid.size(); ++j) {
      dataFile << grid[j] << " " << U[j] << " "
               << std::sin(2 * std::numbers::pi * (grid[j] - a * i * dt))
               << "\n";
    }
    dataFile << "\n\n"; // Separate time steps
  }
  dataFile.close();

  // Create the GNUplot script
  std::ofstream scriptFile("gp_script");
  if (!scriptFile) {
    std::cerr << "Error creating GNUplot script.\n";
    return 1;
  }

  scriptFile << "set terminal qt\n";
  scriptFile << "set xlabel 'x'\n";
  scriptFile << "set ylabel 'u(x,t)'\n";
  scriptFile << "set xrange [" << west << ":" << east << "]\n";
  scriptFile << "set yrange [-1.5:1.5]\n";
  scriptFile << "set grid\n";
  scriptFile << "do for [i=0:" << steps - 1 << "] {\n";
  scriptFile
      << "    plot 'results.dat' index i using 1:2 with linespoints title "
         "sprintf('t = %.2f', i*"
      << dt << " + " << dt
      << "), "
         "'results.dat' index i using 1:3 with lines title 'Exact Solution'\n";
  scriptFile << "    pause 0.1\n";
  scriptFile << "}\n";

  scriptFile.close();

  // Run GNUplot with the script
```

Program's overview

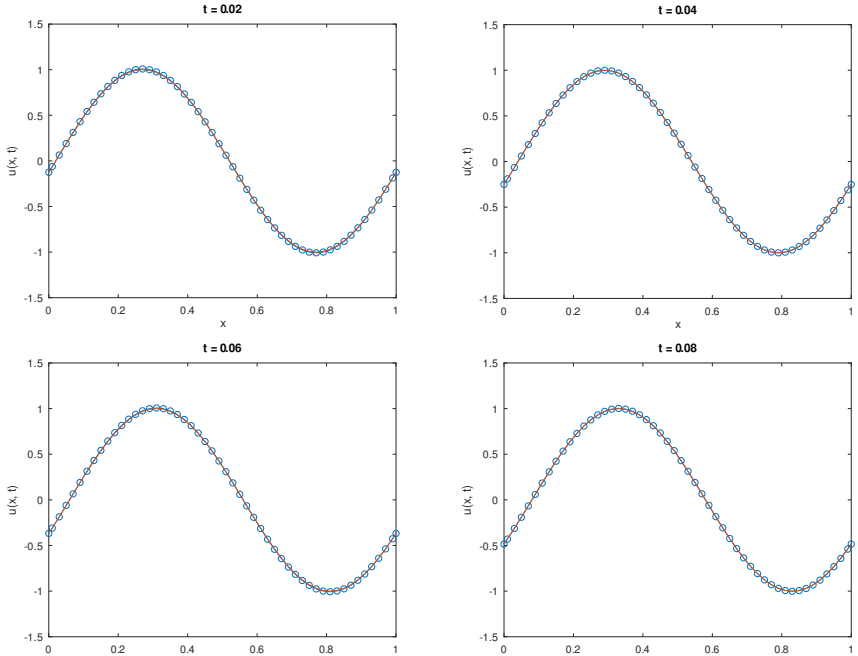
$$I \in \mathbb{R}^{(m+1)\times(m+2)}.$$

$$D \in \mathbb{R}^{(m+2)\times(m+1)}.$$

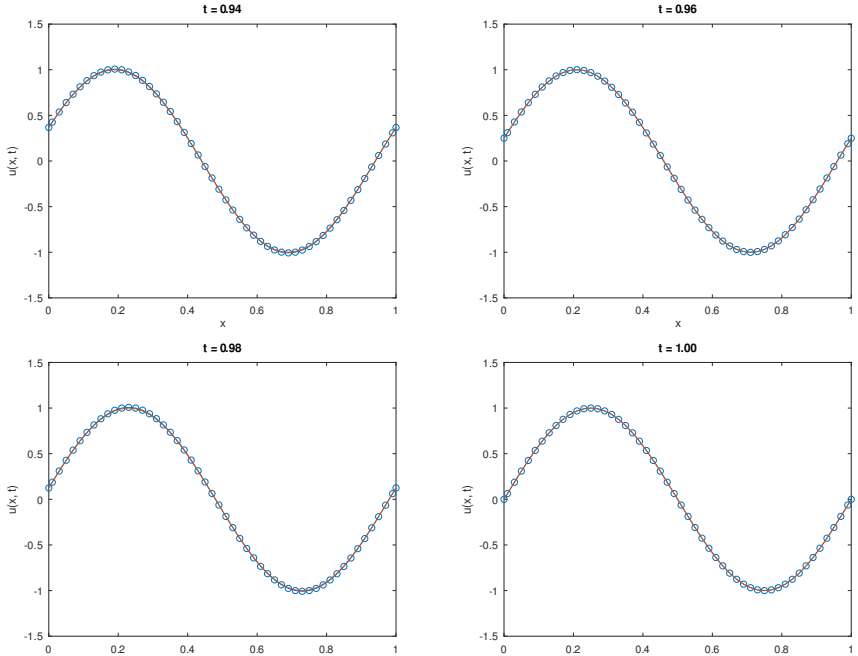
`interpol div`

- The domain $[0,1]$ is divided with 50 cells, $\Delta x = 0.02$.
- The CFL condition ensure that stability for explicit schemes.
- We use the divergence operator of order 2.
- We use the 1D interpolation operator of order 2.
- The initial condition is $u\left(x,0\right) = \sin\left(2\pi x\right)$.
- We modify the first and last row for the divergence operator in order to impose the periodic boundary conditions.
- The spatial operator mixes the divergence and interpolation operators for approximate $-a\partial_x$.
- The factor 2 fixes the scale in the staggered grid.

Results



Results



References

- Books

References

- Articles

References

- Websites

asrinivasan0709. *ICIAM2023*. URL: https://github.com/asrinivasan0709/ICIAM2023 (visited on 03/13/2025).