Solving the 2D Allen-Cahn Equation Using Mimetic Differences

Tyler Collins and Miguel A. Dumett

May 6, 2025

Publication Number: CSRCR2025-01

Computational Science & Engineering Faculty and Students Research Articles

Database Powered by the Computational Science Research Center Computing Group & Visualization Lab

COMPUTATIONAL SCIENCE & ENGINEERING

SAN DIEGO STATE UNIVERSITY

Computational Science Research Center
College of Sciences
5500 Campanile Drive
San Diego, CA 92182-1245
(619) 594-3430

Solving the 2D Allen-Cahn Equation Using Mimetic Differences

Tyler Collins* and Miguel A. Dumett †‡

May 6, 2025

Abstract

This report presents a numerical study of the two-dimensional Allen–Cahn equation using a mimetic finite difference scheme implemented in MATLAB. The method leverages the MOLE library to discretize spatial operators in a way that preserves key vector calculus identities and allows clean enforcement of boundary conditions.

Two test cases were considered: a manufactured solution with a forcing term to evaluate convergence, and an unforced case to observe natural phase evolution. The scheme achieved second-order convergence in the L_2, H^1, and L_∞ norms, validating the accuracy of the mimetic discretization and semi-implicit time integration. Results closely matched published benchmarks, confirming the method's reliability. In the unforced case, the scalar order parameter evolved toward stable binary phases, consistent with the energy-minimizing dynamics of the Allen–Cahn model.

The study demonstrates that mimetic methods provide a robust and interpretable framework for solving phase-field equations on structured grids and lays the groundwork for future extensions to more complex models.

*Computational Science PhD Program at San Diego State University (tcollins7472@sdsu.edu).

†Editor: Jose E. Castillo

‡Computational Science Research Center at San Diego State University (mdumett@sdsu.edu).

1 Introduction

Phase field modeling is a computational approach crucial to engineering and material sciences and is widely used in fracture mechanics, fluid-structure interactions, and multiphase flow [1, 2]. The approach is rooted in van der Waals' diffuse interface theory and was later formalized through the Ginzburg–Landau and Cahn–Hilliard equations, the latter introducing a conserved energy minimization scheme suitable for modeling mass-conserving processes such as phase separation [3]. To simplify the Cahn–Hilliard model, the Allen–Cahn (AC) equation was introduced but at the explicit cost of losing mass conservation [4]. However, the AC equation still reliably captures interface evolution and phase transitions [5]. While conservation is not inherent to the equation, implicit numerical schemes have been developed to control energy dissipation and mitigate mass loss [6, 7, 8].

The AC equation is derived as the gradient flow of the Ginzburg–Landau energy functional and consists of two terms: a linear diffusion term controlling interfacial energy and a nonlinear reaction term driving phase separation [5, 6]. It describes interface dynamics using a continuous scalar order parameter that varies smoothly across phase boundaries and eliminates the need for direct interface tracking. It is commonly applied in microstructural evolution, interface mobility, and pattern formation, and is particularly useful for modeling grain boundary dynamics and crack propagation in materials. Unlike sharp interface methods, the phase-field approach eliminates the need for interface tracking or remeshing, making it well suited to numerical methods such as finite differences, finite elements, and mimetic schemes.

Traditional finite difference methods do not always preserve key physical or geometric properties of the system, whereas mimetic differences can better preserve the system during discretization [9]. Mimetic operators reproduce discrete analogs of vector calculus identities, such as gradient, divergence, and Laplacian, and offer improved stability, convergence, and energy behavior [10]. Additionally, Neumann and Robin boundary conditions are handled more cleanly and do not require any sort of approximation. These attributes are beneficial when modeling diffuse interfaces like those in phase-field models [2].

The objectives of this study are to implement a 2D mimetic finite difference scheme using the Mimetic Operators Library Enhanced (MOLE) to solve the AC equation, validate the implementation by comparing numerical results to an analytical manufactured solution with a forcing term, analyze

the convergence behavior of the solution under mesh refinement, and compare results to existing literature.

2 Mathematical Formulation

2.1 Allen–Cahn Equation

The AC equation describes the time evolution of a non-conserved scalar order parameter $u(x, y, t)$, which distinguishes between phases of a material. In two dimensions, the equation takes the form:

$$\frac{\partial u}{\partial t} = \varepsilon^2 \Delta u - f(u) + g(x, y, t), \tag{1}$$

where:

- $\varepsilon > 0$ is a small parameter related to interfacial thickness.
- Δ is the Laplacian operator (representing isotropic diffusion).
- $f(u) = u^3 - u$ is the derivative of a double-well potential: $F(u) = \frac{1}{4}(u^2 - 1)^2$.
- $g(x, y, t)$ is an optional forcing term.

In the absence of external forcing, the equation simplifies to:

$$\frac{\partial u}{\partial t} = \varepsilon^2 \Delta u - f(u) \tag{2}$$

This equation is derived from the L^2-gradient flow of the Ginzburg–Landau energy functional:

$$E(u) = \int_\Omega \left(\frac{\varepsilon^2}{2} |\nabla u|^2 + F(u) \right) d\Omega \tag{3}$$

With monotonic energy dissipation:

$$\frac{dE(u(t))}{dt} = -\int_\Omega |u_t|^2 \, d\Omega \leq 0 \tag{4}$$

Monotonic dissipation ensures the system minimizes energy over time and reflects thermodynamic principles [6].

2.2 Boundary and Initial Conditions

The problem is defined on a square domain, $(x, y) \in [0, 2\pi]^2$, where homogenous Dirichlet boundary conditions are applied, $u(x, y, t) = 0$. The initial phase field is defined as:

$$u(x, y, 0) = 0.05 \sin(x) \sin(y). \tag{5}$$

This initial perturbation from equilibrium drives a sinusoidal response in the phase field. Without the forcing term this evolves into steady state phase configuration. This initial condition coupled with the forcing term allows for a manufactured analytical solution that we can compare against. The forcing term takes the form:

$$g(x, y, t) = 0.05(2\varepsilon^2 - 1.1)e^{-0.1t} \sin(x) \sin(y) + \left(0.05e^{-0.1t} \sin(x) \sin(y)\right)^3 \tag{6}$$

With the exact solution:

$$u(x, y, t) = 0.05e^{-0.1t} \sin(x) \sin(y) \tag{7}$$

3 Mimetic Differences Approach

3.1 Mimetic Discretization Overview

In this section we discuss relevant mimetic methods using the MOLE library with MATLAB to preserve calculus identities at the discrete level, and describe the mimetic methods approach [11]. Gradient and divergence operators that make up the Laplacian are defined as `grad2D` and `div2D`. However, MOLE pre-constructs the Laplacian with the `lap2D(k, m, dx, n, dy)` function, where `k` is the order of accuracy, `m` and `n` are the number of nodes in the `x` and `y` direction, and `dx` and `dy` are the step sizes in the `x` and `y` direction. Furthermore, to apply Dirichlet boundary conditions the `robinBC2D(k, m, dx, n, dy, a, b)` function, where the parameters are the same as `lap2D` with the exception of, `a`, the Dirichlet coefficient, and, `b`, the Neumann coefficient.

3.2 Discretized Formulation

Spatial discretization is performed using mimetic operators on a staggered, cell-centered grid. The time-stepping scheme follows a semi-implicit Euler

method:

$$\left(I - \Delta t \varepsilon^2 L\right) u^{n+1} = u^n + \Delta t \left(u^n - (u^n)^3 + g^n\right) \tag{8}$$

Where u^n and u^{n+1} represent the numerical solution at time steps t^n and $t^{n+1} = t^n + \Delta t$, respectively. The operator L is the mimetic Laplacian, which discretizes the diffusion operator Δ in space. The expression $\varepsilon^2 L u^{n+1}$ corresponds to the linear diffusion term and is treated implicitly for stability. The nonlinear reaction term is represented by $u^n - (u^n)^3$ and is treated explicitly to avoid solving a nonlinear system. The function g^n is an optional forcing term evaluated at time t^n, used in the manufactured solution case to drive the solution toward a known exact form.

3.3 Algorithm Summary

The algorithm steps in summary:

1. Initialize grid, Laplacian, and boundary matrices.
2. Decompose the implicit system matrix (LU factorization).
3. Loop over time:
    - Assemble RHS with nonlinear term and optional forcing.
    - Solve implicit system for next time step.
    - Unforced solution: store and plot snapshots at intervals.
4. Forced solution: calculate error norms and convergence rate.

4 Implementation

The simulation was implemented in MATLAB using the MOLE library [11]. The parameters used are consistent with the literature [6]. The computational domain was defined on $[0, 2\pi]^2$, with the interfacial thickness parameter ε set to 0.1. Different spatial resolutions were used depending on the test case: simulations with a forcing term used grid sizes of $m = 10$, 20, and 40 to evaluate convergence; simulations without a forcing term used a fixed resolution of $m = 40$. The time step was chosen as $\Delta t = 0.1/m^2$ for the forced case to ensure stability under refinement, and $\Delta t = 0.01$ for the unforced scenario. In the forced case final time $T = 1$, and in the unforced case

final times were run with $T = 1.0$, 2.5, 5.0, and 10.0. Spatial operators were implemented using the MOLE library, which provided mimetic discretizations through functions such as `lap2D` for the Laplacian and `robinBC2D` to enforce homogeneous Dirichlet boundary conditions. The mimetic Laplacian incorporated appropriate boundary corrections consistent with the mimetic framework. To enhance efficiency, the system matrix was factorized once using LU decomposition and reused during each time step of the semi-implicit scheme.

5 Results and Discussion

5.1 With Forcing Term (Manufactured Solution)

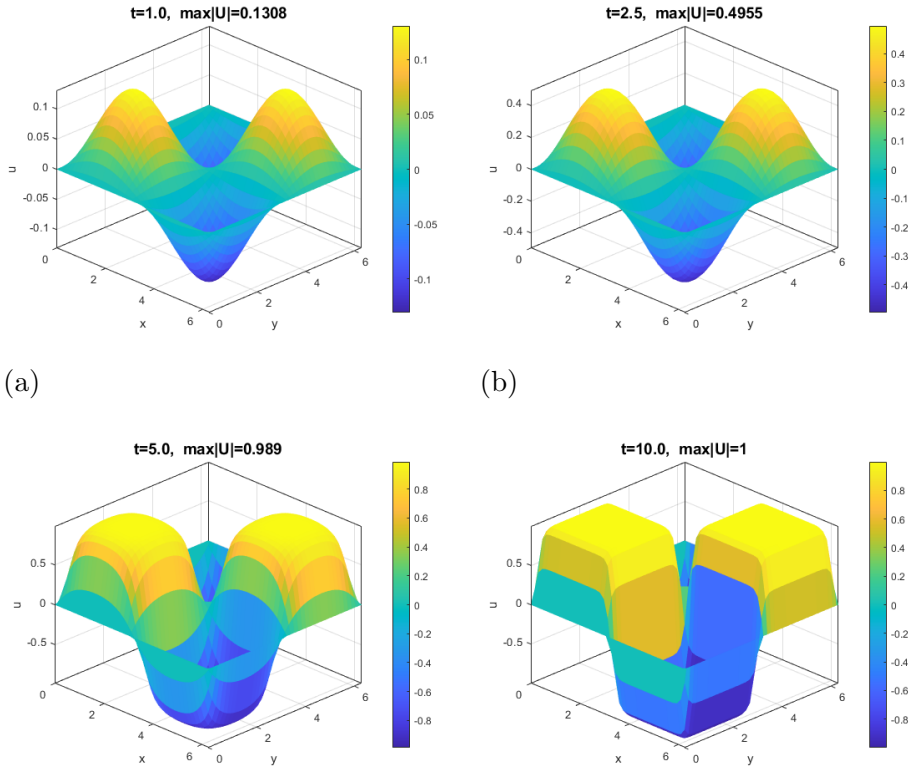
m	h	L_2-error	Rate	H^1-error	Rate	L_∞-error	Rate
10	0.628	3.43E-05	-	5.31E-05	-	1.08E-05	-
20	0.314	7.38E-06	2.22	1.21E-05	2.13	2.21E-06	2.29
40	0.157	1.70E-06	2.11	2.88E-06	2.07	5.35E-07	2.05

Table 1: Computed error norms and convergence rates for manufactured solution.

Running the algorithm with the forcing term from equation (6) and comparing our results with the exact solution from equation (7), we computed the L_2, H^1, L_∞ error and convergence rates, reported in Table 1. We observed error rates from five to seven orders of accuracy, and incremental reductions as mesh sizes increase. The rates of convergence achieved second-order accuracy in space. Although fourth-order spatial operators were used, the overall convergence is governed by the first-order time-stepping and explicit treatment of the nonlinear term, resulting in a second-order global convergence rate. The error results are consistent with Poochinapan *et al.* using a three-level linearized compact difference scheme [6].

5.2 Without Forcing Term (Natural Evolution)

Running the algorithm under unforced conditions with the initial sinusoidal perturbation from equation (5) produced the natural evolution of concentrations shown in Figure 1. At early times, the solution remains diffuse, but

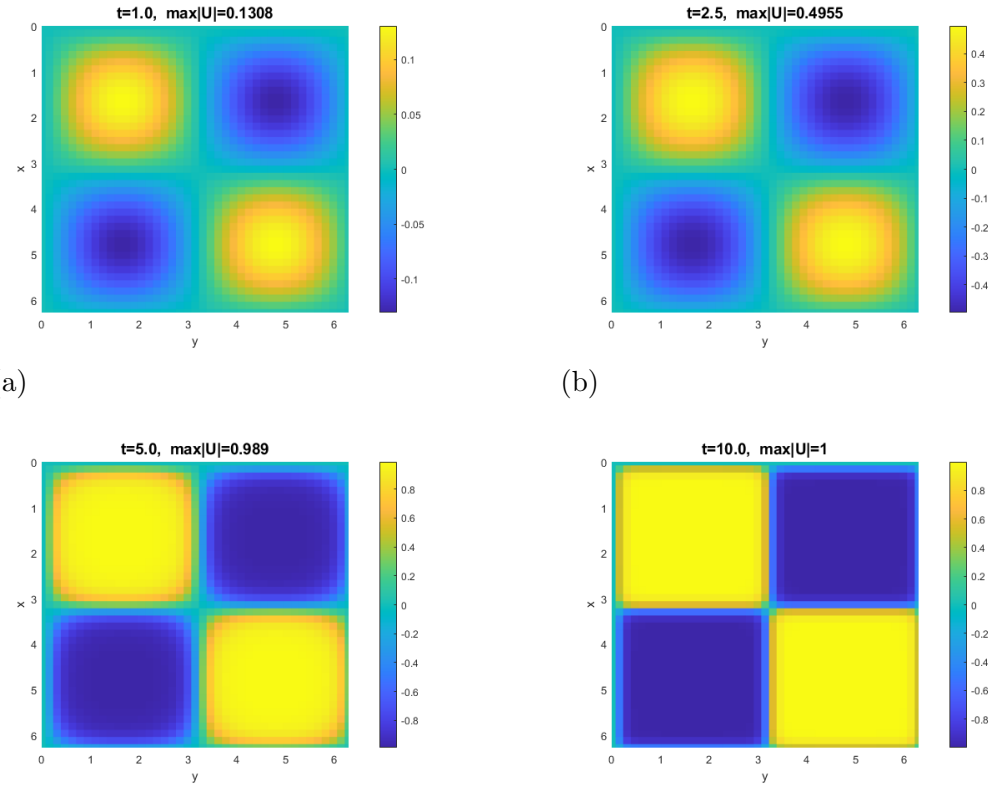


(a) (b)

(c) (d)

Figure 1: Concentration, u, at times a) 1.0, b) 2.5, c) 5.0, and d) 10.0.

as time progresses the interfacial boundaries become sharper, as illustrated in Figure 2. The maximum absolute concentration steadily increases from $u_{\max} = 0.1308$ at $T = 1$ to $u_{\max} = 0.4955$ at $T = 2.5$, then approaches $u_{\max} = 0.989$ at $T = 5$, and saturates to $u_{\max} = 1.0$ by $T = 10$. These observations are consistent with phase separation dynamics governed by the AC equation, where the system gradually minimizes its energy by sharpening interfaces and evolving toward a stable binary phase configuration. Additionally, these results are consistent with the work in Example 3 from Poochinapan *et al.* [6].



(a) (b)

(c) (d)

Figure 2: Top-down view of Figure 1 a), b), c) and d) results showing interface sharpening.

6 Conclusion

This study implemented a mimetic finite difference scheme to solve the 2D AC equation using MATLAB and the MOLE library. The method successfully preserved differential operator identities and incorporated boundary conditions in a stable manner. Simulations using a manufactured solution with a forcing term demonstrated second-order convergence in the L_2, H^1, and L_∞ norms, validating the spatial discretization and temporal integration. These results were in strong agreement with benchmark values reported by Poochinapan *et al.*, despite using a simpler semi-implicit Euler scheme. In the unforced case, the solution exhibited natural phase separation behavior, and the scalar order parameter evolved toward saturated binary phases. The

interface sharpening and energy minimization over time was consistent with the physical interpretation of the AC model.

Overall, the mimetic framework provided a robust, structured, and interpretable approach to solving nonlinear phase-field equations on regular grids. This work establishes a foundation for extending mimetic methods to more complex phase-field systems or higher-order temporal schemes.

References

[1] I. Steinbach, "Phase-field models in materials science," *Model. Simul. Mater. Sci. Eng.*, vol. 17, no. 7, p. 073001, Oct. 2009, doi: 10.1088/0965-0393/17/7/073001.

[2] H. Gomez and K. G. Van Der Zee, "Computational Phase-Field Modeling," in *Encyclopedia of Computational Mechanics Second Edition*, 1st ed., E. Stein, R. Borst, and T. J. R. Hughes, Eds., Wiley, 2017, pp. 1–35. doi: 10.1002/9781119176817.ecm2118.

[3] J. W. Cahn and J. E. Hilliard, "Free Energy of a Nonuniform System. I. Interfacial Free Energy," *J. Chem. Phys.*, vol. 28, no. 2, pp. 258–267, Feb. 1958, doi: 10.1063/1.1744102.

[4] S. M. Allen and J. W. Cahn, "A microscopic theory for antiphase boundary motion and its application to antiphase domain coarsening," *Acta Metall.*, vol. 27, no. 6, pp. 1085–1095, Jun. 1979, doi: 10.1016/0001-6160(79)90196-2.

[5] L.-Q. Chen, "Phase-Field Models for Microstructure Evolution," *Annu. Rev. Mater. Res.*, vol. 32, no. 1, pp. 113–140, Aug. 2002, doi: 10.1146/annurev.matsci.32.112001.132041.

[6] K. Poochinapan and B. Wongsaijai, "Numerical analysis for solving Allen-Cahn equation in 1D and 2D based on higher-order compact structure-preserving difference scheme," *Appl. Math. Comput.*, vol. 434, p. 127374, Dec. 2022, doi: 10.1016/j.amc.2022.127374.

[7] D. He and K. Pan, "Maximum norm error analysis of an unconditionally stable semi-implicit scheme for multi-dimensional Allen–Cahn equations," *Numer. Methods Partial Differ. Equ.*, vol. 35, no. 3, pp. 955–975, May 2019, doi: 10.1002/num.22333.

[8] J. Shen and X. Yang, "Numerical approximations of Allen-Cahn and Cahn-Hilliard equations," *Discrete Contin. Dyn. Syst. A*, vol. 28, no. 4, pp. 1669–1691, 2010, doi: 10.3934/dcds.2010.28.1669.

[9] J. E. Castillo and M. Yasuda, "A Comparison of Two Matrix Operator Formulations for Mimetic Divergence and Gradient Discretizations."

[10] J. Corbino and J. E. Castillo, "High-order mimetic finite-difference operators satisfying the extended Gauss divergence theorem," *J. Comput. Appl. Math.*, vol. 364, p. 112326, Jan. 2020, doi: 10.1016/j.cam.2019.06.042.

[11] J. Corbino, M. A. Dumett, and J. E. Castillo, "MOLE: Mimetic Operators Library Enhanced," *J. Open Source Softw.*, vol. 9, no. 99, p. 6288, Jul. 2024, doi: 10.21105/joss.06288.

A Appendix

A.1 Code

To run the code below the MOLE library must be accessible in MATLAB. Steps for installation can be found at https://github.com/csrc-sdsu/mole. When installed, drop the script into the MOLE examples directory. The code natively runs the unforced scenario, however the forced scenario can be enabled by changing `do_forced = true`. Options for plots and videos are enabled by changing the respective values to `true`.

Listing 1: MATLAB Script: `AC2D_fin_v0.m`

```
clearvars;
clc;
close all;
addpath('../../src/matlab')

%% ==== USER CONFIGURATION ====
do_forced = false;      % true = include forcing term, false = unforced
do_plots  = false;      % true = plot snapshots
do_video  = false;      % true = save video

```

```
% ==== Resolution sweep ====
if do_forced
    Mvals = [10, 20, 40];
else
    Mvals = 40;  % just one resolution for unforced
end

% Error variables initialization
E2    = zeros(size(Mvals));
EH1   = zeros(size(Mvals));
Einf  = zeros(size(Mvals));
hs    = 2*pi ./ Mvals;

for k = 1:length(Mvals)
    eps = 0.1; % interfacial width parameter
    k_order = 4; % mimetic order of accuracy
    m = Mvals(k); % number of nodes
    dx = 2*pi / m; % step height
    dy = dx;
    x = [0, dx/2 : dx : 2*pi - dx/2, 2*pi];
    y = x;
    [X, Y] = meshgrid(x, y);
    N = numel(X);

    if do_forced
        dt = 0.1 / m^2;
        final_time = 1.0;
        plot_times = [];
        plot_steps = [];
    else
        dt = 0.01;
        final_time = 10.0;
        plot_times  = [1, 2.5, 5, 10];
        plot_steps  = round(plot_times / dt);
    end

    nSteps = round(final_time / dt);

```

```
49      % Build operator
50
51      L2D = lap2D(4, m, dx, m, dy) + robinBC2D(4, m, dx,
            m, dy, 1, 0);
52      A = speye(N) - dt * eps^2 * L2D;
53      LU = decomposition(A, 'lu');
54
55      % Initial condition
56      U = reshape(0.05 * sin(X) .* sin(Y), N, 1);
57
58      % Forcing function
59      forcingTermFunc = @(x,y,t) ...
60          0.05*(2*eps^2 - 1.1)*exp(-0.1*t).*sin(x).*sin(y
                ) + ...
61          (0.05*exp(-0.1*t).*sin(x).*sin(y)).^3;
62
63      % === Setup dual video writers (if enabled) ===
64      if do_video
65          v_topdown = VideoWriter(sprintf('
                allenCahn2D_topdown_t%.1f.mp4', final_time),
                 'MPEG-4');
66          v_rotated = VideoWriter(sprintf('
                allenCahn2D_rotated_t%.1f.mp4', final_time),
                 'MPEG-4');
67          v_topdown.FrameRate = 60;
68          v_rotated.FrameRate = 60;
69          open(v_topdown); open(v_rotated);
70      end
71
72
73      % Time stepping
74      for step = 1:nSteps
75          t = step * dt;
76          fU = U - U.^3;
77          forcing = do_forced * reshape(forcingTermFunc(X
                , Y, t), N, 1);
78          rhs = U + dt * (fU + forcing);
79          U = LU \ rhs; % updating solution
```

```
80
81          % ==== Plot at specific times (unforced only)
                ====
82          if do_plots && ~do_forced && ismember(step,
                plot_steps)
83              Umat = reshape(U, m+2, m+2);
84              maxU = max(abs(Umat(:)));
85              t_now = step * dt;
86
87              % --- Top-down view ---
88              figure;
89              surf(X, Y, Umat, 'EdgeColor', 'none');
90              title(sprintf('Unforced Allen Cahn: t =
                    %.2f, max|U| = %.4g', t_now, maxU));
91              xlabel('x'); ylabel('y'); zlabel('u');
                    colorbar;
92              view(90, 90); axis tight; box on;
93
94              % --- Rotated view ---
95              figure;
96              surf(X, Y, Umat, 'EdgeColor', 'none');
97              title(sprintf('Unforced Allen Cahn: t =
                    %.2f, max|U| = %.4g', t_now, maxU));
98              xlabel('x'); ylabel('y'); zlabel('u');
                    colorbar;
99              view(45, 30); axis tight; box on;
100         end
101
102         % Optional video frames
103         if do_video
104             Umat = reshape(U, m+2, m+2);
105             maxU = max(abs(Umat(:)));
106
107             % --- Top-down view ---
108             fig1 = figure('Visible', 'off');
109             surf(X, Y, Umat, 'EdgeColor','none');
110             ax = gca;
```

```
111             ax.ZLim = [-1 1];              % lock z-axis
                    range
112             ax.XLim = [0 2*pi];
113             ax.YLim = [0 2*pi];
114             ax.CLim = [-1 1];              % lock color
                    range (optional but recommended)
115             title(sprintf('t = %.2f, max|U| = %.4g', t,
                     maxU));
116             xlabel('x'); ylabel('y'); zlabel('u');
                    colorbar;
117             view(90,90); box on;
118             writeVideo(v_topdown, getframe(fig1));
119             close(fig1);
120
121             % -- Rotated view --
122             fig2 = figure('Visible', 'off');
123             surf(X, Y, Umat, 'EdgeColor','none');
124             ax = gca;
125             ax.ZLim = [-1 1];              % lock z-axis
                    range
126             ax.XLim = [0 2*pi];
127             ax.YLim = [0 2*pi];
128             ax.CLim = [-1 1];              % lock color
                    range (optional but recommended)
129             title(sprintf('t = %.2f, max|U| = %.4g', t,
                     maxU));
130             xlabel('x'); ylabel('y'); zlabel('u');
                    colorbar;
131             view(45,30); box on;
132             writeVideo(v_rotated, getframe(fig2));
133             close(fig2);
134         end
135
136     end
137
138     if do_video
139         close(v_topdown);
140         close(v_rotated);
```

```
141     end
142
143     % Optional final plot
144     if do_plots && (do_forced || isempty(plot_steps))
145         Umat = reshape(U, m+2, m+2);
146         maxU = max(abs(Umat(:)));
147         figure;
148         surf(X, Y, Umat, 'EdgeColor', 'none');
149         axis tight;
150         title(sprintf('Final time t=%.2f, max|U| = %.4g
                ', final_time, maxU));
151         xlabel('x'); ylabel('y'); zlabel('u'); colorbar
                ;
152         view(90, 90); drawnow;
153     end
154
155     % === Error norms (forced only) ===
156     if do_forced
157         Unum = reshape(U, m+2, m+2);
158         Uex  = 0.05 * exp(-0.1 * final_time) .* sin(X)
                .* sin(Y);
159         err  = Unum - Uex;
160
161         E2(k)   = sqrt(sum(err(:).^2) * dx * dy);
162         dex     = (circshift(err,[0,-1]) - circshift(
                err,[0,1])) / (2*dx);
163         dey     = (circshift(err,[-1,0]) - circshift(
                err,[1,0])) / (2*dy);
164         Egrad   = sqrt(sum((dex(:).^2 + dey(:).^2)) *
                dx * dy);
165         EH1(k)  = sqrt(E2(k)^2 + Egrad^2);
166         Einf(k) = max(abs(err(:)));
167     end
168 end
169
170 % === Print convergence table (forced only) ===
171 if do_forced
```

```
172     fprintf('   M       h          L2-error    rate    H1-
            error    rate    Linf-error    rate\n');
173     for k = 1:length(Mvals)
174         if k == 1
175             fprintf('%4d  %8.4e  %10.4e     -       %10.4
                    e     -       %10.4e     -\n', ...
176                 Mvals(k), hs(k), E2(k), EH1(k), Einf(k)
                        );
177         else
178             r2    = log(E2(k)/E2(k-1))       / log(hs(k)/
                    hs(k-1));
179             rh1   = log(EH1(k)/EH1(k-1))     / log(hs(k)/
                    hs(k-1));
180             rife = log(Einf(k)/Einf(k-1)) / log(hs(k)/
                    hs(k-1));
181             fprintf('%4d  %8.4e  %10.4e  %5.2f    %10.4e
                     %5.2f    %10.4e  %5.2f\n', ...
182                 Mvals(k), hs(k), E2(k), r2, EH1(k), rh1
                        , Einf(k), rife);
183         end
184     end
185 end
```