Numerical PDE Solver

with an adaptive moving mesh

Ayush Singh

We develop a Python-based numerical solver for partial differential equations (PDEs) using finite difference methods. We first implement the solver on a fixed uniform mesh, addressing key components such as spatial and temporal discretization, stability analysis, and scheme selection for representative equations like the advection-diffusion PDE. After validating the baseline solver, we extend it to incorporate an adaptive moving mesh framework. This adaptive method employs monitor functions and the equidistribution principle to dynamically redistribute grid points, focusing resolution in regions with steep gradients or localized features while maintaining efficiency in smooth regions. Our solver achieves improved accuracy and computational efficiency, with applications spanning fluid dynamics, reaction-diffusion systems, and financial modeling such as option pricing. Through this work, we integrate classical numerical analysis with adaptive meshing techniques to create a flexible and efficient PDE solver for complex, real-world problems.

---

1 Introduction

Partial Differential Equations (PDEs): A partial differential equation is a differential equation involving one dependent variable and multiple independent variables, where the derivatives are partial derivatives. In other words, a PDE relates partial derivatives of an unknown multivariable function. PDEs are ubiquitous in modeling physical processes (heat flow, fluid dynamics), as well as financial processes (e.g. option pricing). For example, the famous *Black-Scholes equation* for European option pricing is a PDE in time τ and underlying price S given (for constant volatility σ, interest rate r, and dividend D) by:

$$\frac{1}{2}\,\sigma^2 S^2 \frac{\partial^2 C}{\partial S^2} + (r - D)\,S \frac{\partial C}{\partial S} - r\,C \;=\; -\frac{\partial C}{\partial \tau}$$

for $0 < \tau < T, S > 0$. Here $C(\tau, S)$ is the option price as a function of time and asset price. This is a *parabolic PDE* (akin to a diffusion equation). Solving such PDEs analytically can be challenging or impossible for complex cases, so we resort to numerical methods.

Numerical Solution of PDEs: One of the most common numerical approaches is the finite difference method (FDM). The core idea is to replace continuous derivatives with approximate differences on a discrete grid. In FDM, we first

discretize the domain of the problem into a grid or mesh (often uniform initially). For a one-dimensional spatial domain $[a, b]$, we divide it into N intervals of size $\Delta x = (b-a)/N$, with grid points $x_i = a + i\,\Delta x$ (for $i = 0, 1, \ldots, N$). If the PDE is time-dependent, we also discretize time into time-steps Δt. The differential operators are then approximated by difference formulas relating values at neighboring grid points. This process converts the PDE into a system of algebraic equations that can be solved step by step in time (a timestepping scheme).

2 Fundamentals: Finite Difference Schemes

Finite Difference Approximations: To illustrate, consider a function $u(x, t)$. A first-order partial derivative in space, u_x, can be approximated by a finite difference. For instance, a forward difference formula for u_x at point x_i is:

$$u_x(x_i) \approx \frac{u(x_{i+1}, t) - u(x_i, t)}{\Delta x}$$

which comes from the definition of derivative (dropping the limit). This formula has a truncation error on the order of $O(\Delta x)$ (first-order accurate). A more accurate symmetric formula is the central difference:

$$u_x(x_i) \approx \frac{u(x_{i+1}, t) - u(x_{i-1}, t)}{2\,\Delta x}$$

which is second-order accurate (error $O((\Delta x)^2)$). Likewise, a second derivative u_{xx} can be approximated by a central difference stencil:

$$u_{xx}(x_i) \approx \frac{u(x_{i+1}, t) - 2\,u(x_i, t) + u(x_{i-1}, t)}{(\Delta x)^2}$$

These approximations allow us to discretize spatial derivatives in the PDE. For example, a diffusion term like $\partial^2 u/\partial x^2$ can be replaced by the above second difference formula.

Time Discretization: For time derivatives, one can use analogous finite differences. A simple forward Euler step for the time derivative u_t is:

$$u_t(t^n) \approx \frac{u^{n+1} - u^n}{\Delta t}$$

where $u^n(x)$ denotes the solution at the n-th time step $t^n = n\,\Delta t$. Using this, an explicit update formula for a time-dependent PDE $u_t = F(u, u_x, u_{xx}, \ldots)$ can be written as:

$$u^{n+1}i = u_i^n + \Delta t\; F(u_i^n, (u_x)_i^n, (u)_i^n, \ldots)$$

with spatial derivatives replaced by finite differences. This yields a time-stepping scheme, advancing the solution from t^n to t^{n+1}. The scheme is called explicit because u^{n+1} is computed directly from known values at time n. In contrast, an implicit scheme (e.g. backward Euler or Crank-Nicolson) would involve u^{n+1} on the right-hand side as well, leading to a system of equations to solve at each step. Implicit schemes are unconditionally stable for many problems (allowing larger Δt), but require solving linear systems, whereas explicit schemes are conditionally stable but straightforward to implement.

Stability Considerations: Choosing appropriate Δx and Δt is crucial. The discretization introduces two types of error: truncation error (from the finite difference approximation) and roundoff error (from finite precision arithmetic). Moreover, the time-stepping scheme can suffer from numerical instability if Δt is too large relative to Δx. For explicit schemes, one often must satisfy a Courant=Friedrichs-Lewy (CFL) condition: essentially, the numerical domain of dependence should cover the PDE's domain of dependence. For example, a simple stability criterion for an advection equation $u_t + c\,u_x = 0$ on a uniform mesh is $c\,\Delta t \leq \Delta x$ (so that the information does not skip over a grid cell in one time-step). In general, exceeding such a limit causes the solution to blow up with oscillations. The necessity of a stability criterion is well-known; practitioners often check the von Neumann stability analysis and CFL condition for their scheme. In practice, a balance is needed: smaller $\Delta x, \Delta t$ improve accuracy but increase computational cost. If an explicit scheme becomes too restrictive, an implicit scheme can be used at the cost of solving equations each step.

Example: Advection-Diffusion Equation: As a concrete example, consider the 1D advection-diffusion equation, which combines a first derivative (advection or convection term) and second derivative (diffusion term):

$$\frac{\partial u}{\partial t} + a\,\frac{\partial u}{\partial x} = D\,\frac{\partial^2 u}{\partial x^2}$$

with constant advection speed a and diffusion coefficient D. This type of PDE can model, for instance, pollutant transport in a pipe with advection velocity a and diffusivity D. We would impose an initial condition $u(0, x) = u_0(x)$ (e.g. an initial concentration profile) and appropriate boundary conditions (which might be fixed values at domain ends or periodic, depending on the physical scenario). To solve it numerically:

- *Spatial discretization:* Replace u_x and u_{xx} with finite differences on grid $x_0, \ldots, x_N$. For stability in advection-dominated problems, a upwind difference is usually employed for the u_x term. For example, if $a > 0$ (flow

to the right), use a backward difference $u_x \approx (u_i - u_{i-1})/\Delta x$ (which uses the upwind value from where the characteristic comes) – this helps damping unphysical oscillations in convection-dominated cases. The diffusion term can be handled with the central second difference above.

- *Time stepping:* We could use an explicit Euler scheme:

$$u^{n+1}i = u_i^n - a\,\frac{\Delta t}{\Delta x}\left(u_i^n - u^n\right) + D\,\frac{\Delta t}{(\Delta x)^2}\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right)$$

Here the second term comes from the upwind approximation of $a\,u_x$ and the third term from the standard diffusion second difference. This scheme will be stable if Δt satisfies both the advection CFL condition ($a\,\Delta t \le \Delta x$) and a diffusion condition ($D\,\Delta t \le \frac{1}{2}(\Delta x)^2$ for the simple explicit method). If these conditions are too restrictive, one could use an implicit or Crank–Nicolson scheme for the diffusion part (since diffusion is often the stiff part requiring small time steps).

Implementation in Python: We would create arrays for u^n and update to u^{n+1} using the above discrete formula inside a time loop. After each time-step, apply boundary conditions (e.g. set u_0^{n+1} and u_N^{n+1} to given boundary values). This yields a simulation of the PDE over time.

This main project of building a Python solver involves understanding the model equation, discretizing it, and ensuring stability and accuracy. Once the base solver on a fixed uniform mesh is working and verified (for example by comparing to known analytical solutions or checking conserved quantities), we can move on to the extension: an adaptive moving mesh to improve efficiency and accuracy.

3 Adaptive Moving Mesh

After completing the main project on a fixed grid, an advanced extension is to incorporate adaptive mesh refinement or movement. The goal is to concentrate computational effort (grid points) where the solution has sharp features or large errors, and conversely use a coarser grid where the solution is smooth. This can dramatically improve accuracy without a proportional increase in computational cost.

Adaptive Grid Concepts: An adaptive grid automatically clusters grid points in regions of high solution gradients or error . In other words, as the solution evolves, the grid points themselves move or the grid refines to “zoom in” on steep or important features. There are two primary strategies for mesh adaptation: h-adaptation and r-adaptation:

- h-adaptation (refinement): The mesh topology changes by adding or removing points (refining or coarsening) in regions based on error indicators. For example, one might split cells that have large solution error or gradient, and merge cells in flat regions . This changes the number of grid points locally.
- r-adaptation (relocation): The total number of grid points is fixed, but they are relocated to change the distribution of mesh density . Essentially the mesh points slide along the domain, clustering in regions of interest and spreading out in others. This is also called a moving mesh method, since the mesh nodes move continuously with time.
- (There is also p-adaptation, which increases the polynomial order of the numerical method instead of changing the mesh, often used in finite element methods , but here we focus on mesh adaptation.)

The extension requested is specifically an adaptive moving mesh, i.e. an r-adaptive approach. We will keep a fixed number of points but allow their positions to evolve over time to track the solution features.

When/Why to Use Moving Mesh: If the solution develops sharp gradients, moving fronts, or localized structures (such as shock waves, boundary layers, or in finance, a kink at an option strike or a moving barrier), a uniform grid either wastes many points in smooth areas or requires an extremely fine mesh everywhere to resolve the sharp feature. Adaptive meshing concentrates resolution where needed. A classic example is a traveling shock or steep wave: rather than use thousands of points uniformly, a moving mesh method will move a cluster of points along with the shock, maintaining resolution around it. This often yields better accuracy for the same number of points.

Monitor Function and Equidistribution: The central idea in r-adaptive methods is to define a monitor function $M(x,t)$ that measures where the mesh should be dense. Typically, $M(x,t)$ is larger where the solution has large gradients or error. For example, one simple choice is $M(x,t) = 1 + \alpha\,|u_x(x,t)|$ (with some coefficient α) so that regions of high slope have a high monitor value. Other choices might use the second derivative or an estimate of the local error. This M acts like a "density" of mesh points. We then require the mesh to equidistribute this monitor function: each grid cell should contain approximately equal amount of the monitor. Formally, if the physical domain is $[a,b]$ and we have N subintervals (with $N+1$ grid points), equidistribution means:

$$\int_{x_{i-1}}^{x_i} M(x,t)\,dx \;\approx\; \frac{1}{N}\int_a^b M(x,t)\,dx$$

for every cell $i = 1, \ldots, N$. In other words, each cell carries an equal share of the total "monitor mass". This principle, first introduced by de Boor in the context of ODE boundary value problems , is the foundation of many moving mesh methods. It ensures that where M is large (large error/gradient), the cells $\Delta x_i = x_i - x_{i-1}$ become small (lots of points clustered), and where M is small, cells can be larger. By equidistributing, we are effectively equalizing the interpolation or truncation error across the domain - no region is over-refined or under-resolved relative to the monitor.

Implementing the Moving Mesh: To incorporate this into your solver, follow these general steps:

1. Choose a monitor function $M(x,t)$: Often based on solution gradients or curvature. For example, $M = \sqrt{1 + (u_x)^2}$ is a popular monitor aiming to equidistribute the arc length of the solution curve . You may add a small floor value (like $M = \epsilon + \sqrt{1 + (u_x)^2}$) to avoid M being zero in constant regions. The parameter ϵ prevents cells from becoming too large in absolutely flat regions.

2. Compute the monitor on the current mesh: At each time step (or every few steps), evaluate $M_i \approx M(x_i, t)$ at the current grid points (or on sub-cells).

3. Equidistribute and obtain a new mesh: We need to find new grid point positions x_i^{new} such that $\int_a^{x_i^{\text{new}}} M(x,t)dx = \frac{i}{N}\int_a^b M(x,t)dx$ for each i. Practically, this can be done by numerical integration of M and inversion of the cumulative distribution. For instance, compute the cumulative integral of M over $[a,b]$, then choose each x_i^{new} where the cumulative integral equals i/N of the total. This approach ensures each subinterval has equal monitor mass. (This is analogous to constructing a mesh via the inverse CDF method where M plays the role of a density function.) There are libraries and algorithms for this; one could also use a simple iterative method or rootfinding to place points. Classic moving mesh algorithms like de Boor's method or the moving mesh PDE approach essentially achieve this equidistribution, possibly with some smoothing. In fact, many methods solve a so-called moving mesh PDE (MMPDE) which gradually moves the mesh toward the equidistributed state . But conceptually, you can directly reposition the points by solving the equidistribution equation at each adapt step.

4. Interpolate the solution onto the new mesh: After moving the nodes, the solution u defined on the old grid x_i^{old} must be interpolated to the new grid x_i^{new}. Typically linear interpolation is used (higher-order interpolation could be used but might introduce oscillations if not careful). Since the mesh might move every time step, it's important that interpolation is accurate and ideally conservative (especially if u represents a conserved quantity like mass). Using an interpolation at least as accurate as the original scheme is recommended . After interpolation, we have $u(x_i^{new}, t)$ values ready for the next time step.

5. Proceed with time integration: Now use the PDE finite difference update on this new mesh for the next time step. When computing spatial derivatives on a non-uniform mesh, the finite difference formulas need to be generalized. For example, a central difference on a nonuniform grid: $u_x(x_i) \approx \frac{(x_i - x_{i-1})\, u(x_{i+1}) - (x_{i+1} - x_i)\, u(x_{i-1})}{(x_{i+1} - x_{i-1})/2}$ (one can derive these or use interpolation to a uniform computational space – see below). Another approach is to map the physical domain $[a, b]$ with a moving physical mesh to a fixed computational domain (often taken as $[0, 1]$ for convenience) with a uniform computational mesh. Then the solution can be reformulated in terms of computational coordinates $\xi \in [0, 1]$. The r-adaptive method essentially introduces a time-dependent mapping $x(\xi, t)$ between computational coordinate ξ and physical position x . The PDE in physical coordinates transforms to a PDE in (ξ, t) plus additional terms from mesh motion (or one can solve the physical PDE with a moving mesh equation coupled in). While the full algorithm can become complex (solving a mesh motion PDE along with the original PDE), the simpler view for implementation is the rezoning approach described above: every so often, re-compute a good mesh (by equidistribution) and interpolate.

Illustration of Adaptive vs Uniform Mesh: The figure below demonstrates how an adaptive moving mesh can concentrate points in regions of interest. In this example, the function $u(x)$ has a steep jump near $x = 0.5$. A uniform grid uses points evenly spaced, resolving the jump poorly, whereas an adaptive mesh (with the same total points) clusters densely around $x = 0.5$ to capture the sharp change with higher resolution.

Uniform vs adaptive mesh point distribution for a function with a steep gradient. The adaptive moving mesh uses the same number of points but allocates them according to solution features (here, clustering around the sharp jump) to better resolve the behavior.

3.1 Practical Considerations

1. *How often to adapt:* Adapting the mesh at every time step provides the most responsive tracking of features but also incurs more overhead (interpolation every step). In many cases, adapting every few time steps or when an error indicator exceeds a threshold can be sufficient and more efficient.

2. *Mesh movement limits:* To avoid overly distortive mesh configurations, some algorithms impose smoothing or limit how fast points can move. This prevents tangling or extremely skewed cells. In one strategy, one might only move a fraction of the way to the equidistributed mesh each time step, rather than full repositioning. More advanced methods solve a parabolic mesh relaxation equation (the MMPDE) to move nodes gradually.

3. *Boundary points:* Typically the domain boundaries remain fixed ($x_0 = a$, $x_N = b$ fixed in time) while interior points move. Ensure the monitor function and equidistribution enforce that no points cross the boundaries or each other (the equidistribution method above inherently preserves ordering of points if done consistently).

4. *Stability and CFL:* If the mesh becomes highly non-uniform, the smallest $\Delta x_{\min}$ may force a very small Δt for stability of explicit schemes (because CFL involves the smallest cell). Monitor-based adaptivity can sometimes lead to extremely small cells. If this happens, you might switch to an implicit scheme or impose a lower bound on cell sizes (e.g., add a small ϵ to M as mentioned to avoid Δx going to zero). Always recompute the CFL condition after adapting the mesh.

5. *Verification:* Test the moving mesh code on known solutions. For example, if you know an analytic solution or a symmetry, ensure the adaptive solver still produces correct results. Also, verify that when the solution is smooth or constant, the adaptive mesh remains near-uniform (no spurious clustering).

4 Learning Path

To successfully build this extension, you should:

- Master the basics of PDE numerics: Ensure you are comfortable with your base solver on a fixed mesh. Understand stability criteria (like CFL) and error sources. This is assumed from the main project.

- Learn about mesh adaptation techniques: The theory of adaptive methods is a rich field. Key concepts include the equidistribution principle (as we used above) and error estimation. For a deeper dive, see references like Huang and Russell (2011) "Adaptive Moving Mesh Methods", which provide systematic derivations of mesh equations. Understanding the idea of a monitor function and how it relates to error will guide your choices. The Stack Exchange community and computational science texts have many discussions on implementing moving meshes.

- Plan simple tests: Start with a simpler monitor (e.g., one based on known solution shape or a fixed target distribution) to verify your mesh moving procedure. For instance, if you know the solution will be peaked at a certain location, test that your equidistribution code indeed places more points there. You might initially freeze the solution and just test the mesh generator.

- In Python, you can integrate the monitor using numerical quadrature (e.g., Simpson's rule on each cell) to get the cumulative distribution. Use interpolation (e.g. NumPy's interp or SciPy) to invert it and find new grid points. For interpolation of u onto the new grid, you can use NumPy interpolation for simplicity (linear) or higher-order using SciPy if needed. Ensure edge cases are handled (the first and last points remain at boundaries, etc.).

- Combine solver and mesh movement: Once the mesh movement is working, integrate it with the time-stepping loop. A possible algorithm:

    1. Compute u^{n+1} on the current mesh (one time step of PDE solver).
    2. Evaluate monitor $M(x, t^{n+1})$ using u^{n+1}.
    3. Compute new grid x_i^{new} by equidistributing M.
    4. Interpolate u^{n+1} onto x_i^{new} to get the new representation of solution.
    5. Continue to next step with this new mesh and u.

Optionally, you might not adapt every single step - you could do several PDE time steps, then an adapt step. This can sometimes reduce interpolation overhead if features aren't moving too fast.

Verification and Benefits: An adaptive moving mesh should yield a solution of comparable accuracy to a fine uniform mesh, but with far fewer points. For example, a moving mesh was successfully used to resolve sharp moving fronts in reaction-diffusion systems with high accuracy, where a uniform mesh would

have required many more points. In computational fluid dynamics, moving mesh methods have tackled shock tracking and boundary layer resolution effectively. In finance, adaptive meshes have been applied to option pricing (e.g., focusing grid points near the option's strike or barrier where payoff curvature is high) to improve computational efficiency.

Keep in mind that adding adaptivity introduces complexity: the code is more elaborate and there are more things that could go wrong (interpolation errors, mesh tangling, etc.). Thus, always test your adaptive solver against the baseline fixed-mesh solver for problems where you know the answer. A wellimplemented adaptive mesh will equidistribute the error, giving you a more uniform error distribution and often a smaller maximum error than a uniform grid with the same number of points.

5 Conclusion and Further Resources

In summary, to build this project you started with formulating the PDE and a finite difference scheme in Python, covering all necessary math from derivative approximations to stability criteria. With the adaptive moving mesh extension, you introduced an intelligent mesh that reallocates itself according to the solution's needs, guided by the equidistribution principle and monitor functions. This document has provided definitions, formulas, and step-by-step guidance, essentially serving as a comprehensive reading resource for understanding and implementing the project.

For further reading, the literature on adaptive meshes is extensive. You might consult Adaptive Moving Mesh Methods by Huang and Russell for a deep theoretical foundation , or research articles on specific applications (e.g., adaptive mesh for Black-Scholes equations or for fluid dynamics shocks). The computational science community (e.g., Stack Exchange discussions) is also a valuable source of practical tips and insights from people who have implemented such methods. By mastering these concepts and techniques, you will be equipped to tackle the project and successfully implement an adaptive moving mesh solver in Python, achieving efficient and accurate solutions for challenging PDE problems.