

NUMERICAL SOLUTION OF PARTIAL DIFFERENTIAL EQUATIONS USING FINITE ELEMENT METHOD (FEM) IMPLEMENTED IN PYTHON FOR SCIENTIFIC COMPUTING APPLICATIONS

Abdul Rehman Nangraj^{*1}, Sabira², Ghulam Yameen Mallah³

^{*1}Lecturer Institute of Mathematics and computer Science. University of Sindh Jamshoro

²Assistant Professor, Department of Education and Literacy, Govt. Girls Degree College Kotri

³Assistant Professor, Department of Basic Science and Related Studies, Quaid-e-Awam University of Engineering Science and Technology Nawabshah Sindh

^{*1}nangraj@usindh.edu.pk, ²sabirakhurram@gmail.com, ³gh.yameen@quest.edu.pk

DOI: <https://doi.org/10.5281/zenodo.19874921>

Keywords

Finite Element Method (FEM), Partial Differential Equations (PDEs), Python Scientific Computing, FEniCSx, Firedrake, PETSc, High-Performance Computing, Mesh Generation, Automatic Differentiation, Matrix-Free Methods, GPU Computing, Physics-Informed Neural Networks (PINNs), Exascale Computing, Variational Formulation, Sparse Linear Algebra

Article History

Received: 01 January 2026

Accepted: 10 March 2026

Published: 28 March 2026

Copyright @Author

Corresponding Author: *

Abdul Rehman Nangraj

Abstract

The Finite Element Method (FEM) remains a cornerstone for solving partial differential equations (PDEs) in scientific and engineering applications, yet its implementation has historically been confined to low-level compiled languages. This paper presents a comprehensive examination of the modern paradigm shift toward Python-based FEM frameworks, which serve as high-level orchestration layers binding sophisticated geometry kernels, meshing libraries, and parallel linear algebra backends into coherent research workflows. We systematically analyze the mathematical foundations of FEM, including the transition from strong to weak formulations via the Galerkin method, and investigate the ecosystem of Python libraries including FEniCSx, Firedrake, FElupe, and SfePy. Performance benchmarks demonstrate that while pure Python execution incurs significant overhead relative to compiled languages, the offloading of computationally intensive operations to optimized backends such as PETSc, Trilinos, and GPU-accelerated matrix-free solvers reduces Python overhead to a negligible fraction of total runtime. Key findings reveal that mesh generation remains the primary bottleneck in FEM workflows, with element quality directly influencing numerical accuracy through Jacobian determinants and aspect ratios. Emerging frontiers including automatic differentiation, differentiable physics frameworks like JAX-FEM, physics-informed neural networks (PINNs), and neuromorphic hardware implementations on platforms such as Intel's Loihi 2 demonstrate the evolving landscape beyond traditional FEM. This analysis concludes that the fusion of Python's high-level productivity with low-level computational efficiency ensures FEM will remain indispensable for exascale scientific computing.

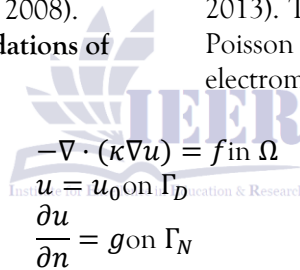
1. INTRODUCTION

The computational landscape for scientific and engineering discovery has undergone a profound transformation, driven by the increasing

complexity of physical models and the diversification of high-performance computing

(HPC) architectures. At the heart of this evolution lies the numerical solution of partial differential equations (PDEs), which govern phenomena ranging from structural mechanics and fluid flow to electromagnetic wave propagation and mass transport (Maas *et al.*, 2017). Among the methodologies available to discretize and solve these equations, the Finite Element Method (FEM) remains a standard for its versatility in handling complex, unstructured geometries and its rigorous mathematical foundation (Niiranen, 2021). Historically, the development of FEM software was the exclusive domain of low-level compiled languages like Fortran and C++ due to the extreme computational demands of large-scale simulations (Besson *et al.*, 1998). However, the modern paradigm has shifted toward Python as the primary interface for scientific computing, serving as a high-level "glue" that binds sophisticated geometry kernels, meshing libraries, and parallel linear algebra backends into coherent research workflows (Vallisneri & Babak, 2008).

2. Mathematical and Theoretical Foundations of Finite Element Discretization



$$\begin{aligned} -\nabla \cdot (\kappa \nabla u) &= f \text{ in } \Omega \\ u &= u_0 \text{ on } \Gamma_D \\ \frac{\partial u}{\partial n} &= g \text{ on } \Gamma_N \end{aligned}$$

Here, u is the unknown field, κ is a material coefficient, f is a source term, and Γ_D and Γ_N represent Dirichlet (essential) and Neumann (natural) boundary conditions, respectively. While the strong form requires u to possess sufficient smoothness (typically twice differentiable), the finite element method (FEM) instead relies on a weak or variational formulation, which relaxes

The finite element method is fundamentally a procedure for approximating the unknown solution of a boundary value problem by subdividing the domain into smaller, simpler parts called finite elements (Bueler, 2025). This subdivision, or space discretization, allows for the representation of complex geometries through a mesh of points, or nodes, which define the numerical domain. The primary advantage of this approach lies in its ability to capture local effects and represent dissimilar material properties across the domain with high fidelity (Figueira *et al.*, 2025).

2.1 The Strong Form and the Variational Principle

In the context of mathematical physics, a partial differential equation (PDE) is typically first presented in its strong form, consisting of the governing differential equation together with a set of boundary conditions (Larson & Bengzon, 2013). To illustrate, consider the classical elliptic Poisson equation, which arises in electromagnetics, heat transfer, and electrostatics:

differentiability requirements and is more suitable for numerical approximation (Ciarlet, 2002).

The weak form is obtained using the Galerkin method. The governing equation is multiplied by an arbitrary test function v belonging to a suitable Hilbert space V_0 , where $v = 0$ on Γ_D . Integrating over the domain Ω and applying Green's first identity (integration by parts) yields the variational formulation: find $u \in V$ such that

$$\int_{\Omega} \kappa \nabla u \cdot \nabla v \, dx = \int_{\Omega} f v \, dx + \int_{\Gamma_N} g v \, ds \quad \forall v \in V_0$$

This weak formulation reduces approximation error by embedding the PDE into an integral framework that enables flexible selection of trial and test function spaces.

In modern computational frameworks such as FEniCSx and Firedrake, this variational structure is expressed using the Unified Form Language (UFL), which provides a symbolic representation closely aligned with the mathematical derivation,

enabling a near one-to-one translation from theoretical formulation to executable code (Reddy, 2019).

2.2 Discretization and Function Spaces

The continuous variational problem is transformed into a discrete algebraic system by choosing finite-dimensional subspaces V_h

$\subset V$. These spaces consist of piecewise polynomial functions defined over the mesh elements. The choice of basic functions ranging from linear Lagrange elements to high-order hierarchical Lobatto basis determines the accuracy and convergence rate of the method (Cimrman *et al.*, 2014).

Table 1: Common Finite Element Types and Their Applications

Element Type	Basis Functions	Typical Applications
Lagrange (P1, P2)	Globally continuous piecewise polynomials (Alnæs <i>et al.</i> , 2014; Baratta <i>et al.</i> , 2023).	Standard structural and thermal analysis (Alnæs <i>et al.</i> , 2015; Baratta <i>et al.</i> , 2023).
Crouzeix-Raviart	Non-conforming elements with continuity at midpoints.	Fluid dynamics and topology optimization (Jia <i>et al.</i> , 2024).
B-bar elements	Projected dilatational strains on piecewise constant spaces (Bueler, 2025).	Incompressible materials (Poisson's ratio ≈ 0.5) (Bueler, 2025).
Lobatto basis	Hierarchical polynomials for tensor-product elements (Cimrman <i>et al.</i> , 2014).	High-order approximations in spectral-like FEM (Cimrman <i>et al.</i> , 2014).

The discretization process results in a sparse system of linear equations, $Ku=FKu=F$, where KK is the stiffness matrix and FF is the load vector. The sparsity of KK is a critical computational feature; since the basic functions have small support, most entries in the matrix are zero. This sparsity is what allows the finite element method (FEM) to scale to millions of degrees of freedom on modern hardware (Balay *et al.*, 2026).

3. The Ecosystem of Python FEM Libraries

Python's dominance in scientific computing is not due to its execution speed but its ability to orchestrate complex software stacks. Modern FEM libraries in Python use sophisticated code generation to produce high-performance C or CUDA kernels while providing a productive high-level API (Rathgeber *et al.*, 2016).

3.1 FEniCS and the Evolution to FEniCSx

The FEniCS Project has established itself as a cornerstone of automated FEM research. It provides a comprehensive platform for solving PDEs by automatically translating UFL symbolic forms into efficient machine code. The latest

iteration, FEniCSx (including DOLFINx), has been redesigned for modularity and performance, integrating seamlessly with modern Python tools like NumPy and Numba without sacrificing the speed of C++ implementations (Alnæs *et al.*, 2014).

FEniCSx components work in a tightly coupled pipeline:

1. **Basix**: Manages the definition of finite element basis functions.
2. **UFL**: Provides the symbolic language for variational formulations.
3. **FFCx**: Acts as the just-in-time (JIT) compiler that generates element-level kernels.
4. **DOLFINx**: Executes these kernels across the mesh and manages global assembly (Baratta *et al.*, 2023).

This architecture allows for significant flexibility. For instance, in frequency-domain vibro-acoustic simulations, FEniCSx has been used to bridge non-conformal mesh interfaces using specialized interpolation matrices, achieving performance levels comparable to major commercial software (Alnæs *et al.*, 2015).

3.2 Firedrake: Composability and Performance Portability

Firedrake offers a similar high-level interface to FEniCS but adopts a different architectural philosophy focused on the "composition of abstractions". It decouples the FEM-specific logic from the parallel execution layer, using a runtime-only implementation centered on the PyOP2 interface. This separation of concerns allows computer scientists to optimize the parallel loop execution while numerical analysts focus on element formulations (Rathgeber *et al.*, 2016). Firedrake is particularly recognized for its support of:

- **High-order elements:** Utilizing sum factorization and vectorization to minimize the computational footprint.
- **Block-structured solvers:** Providing native support for composable physics-based preconditioners.
- **Custom operators:** Offering an API for operations that fall outside pure variational forms, such as flux-limiters (Ham *et al.*, 2023).

3.3 Specialized Libraries and Toolkits

Beyond general-purpose solvers, the Python ecosystem includes several specialized toolkits tailored for specific engineering domains:

Table 2: Specialized Python FEM Libraries and Their Primary Focus

Library	Primary Focus	Distinguishing Feature
FElupe	Nonlinear continuum mechanics (Dutzler <i>et al.</i> , 2021).	High-level API for hyperelastic solid bodies with automatic differentiation (Dutzler <i>et al.</i> , 2021).
SfePy	General-purpose simple FEM (Cimrman <i>et al.</i> , 2014).	Supports multiscale homogenization and isogeometric analysis (Cimrman <i>et al.</i> , 2014).
FiPy	Finite Volume Method (FVM) (NIST, 2026).	Optimized for phase-field and diffusion-convection problems (NIST, 2026).
FEniTop	Topology optimization (Jia <i>et al.</i> , 2024).	Built on FEniCSx; automates sensitivity analysis and CAD export (Jia <i>et al.</i> , 2024).
PyLith	Geodynamics and seismology (PyLith Team, 2024).	Supports complex fault geometries and tectonic modeling (PyLith Team, 2024).

FElupe is particularly effective for modeling elastomer components, leveraging NumPy-based math to perform efficient element-wise operations. SfePy, on the other hand, excels in scientific research applications where users need a "black-box" solver extended with custom term libraries. It is noteworthy that while FiPy is often discussed alongside FEM libraries, it fundamentally uses the Finite Volume Method (FVM), which lacks the higher-order flexibility inherent to FEM (NIST, 2026).

4. High-Performance Computing Backends and Exascale Readiness

The computational efficiency of Python-based Finite Element Method (FEM) frameworks is largely achieved by delegating computationally intensive tasks—such as sparse matrix assembly and the solution of large-scale linear systems—to highly

optimized, parallel backend libraries (Dalcin *et al.*, 2011). This separation between high-level modeling and low-level numerical execution is fundamental for scalability in modern scientific computing.

At the core of FEM discretization lies the standard algebraic system:

$$Ku = f$$

where K is the global stiffness matrix, u is the unknown solution vector, and f is the external force/load vector.

For nonlinear PDEs, solution procedures typically rely on Newton-type iterations, where each step requires solving:

$$J(u^k) \Delta u^k = -R(u^k)$$

Here, $J(u^k)$ is the Jacobian matrix, $R(u^k)$ is the residual, and Δu^k updates the solution at iteration k .

4.1. Scalable Solvers with PETSc and Trilinos

The Portable, Extensible Toolkit for Scientific Computation (PETSc) provides scalable Krylov subspace solvers for large sparse systems of the form:

$$Ax = b$$

where A represents a sparse operator, x the unknown vector, and b the right-hand side. PETSc enables distributed-memory parallelism and GPU acceleration through abstractions that decouple algorithm design from hardware execution.

Similarly, Trilinos supports modular solver design using components such as Tpetra and Kokkos, enabling portability across CPUs and GPUs without modifying core algorithms.

4.2. Emerging GPU Architectures and Matrix-Free Methods

As high-performance computing advances toward exascale systems, arithmetic intensity becomes a critical bottleneck. To address memory limitations, modern FEM frameworks such as MFEM increasingly adopt matrix-free formulations.

Instead of explicitly forming sparse matrices, the operator is applied directly:

$$y = Av = \mathcal{F}(v)$$

where $\mathcal{F}(v)$ represents the element-wise or operator-level action of the discretized PDE. This approach significantly reduces memory bandwidth requirements and improves GPU utilization.

Matrix-free methods are particularly effective for high-order FEM discretizations, where assembling A would otherwise dominate both memory and computation cost.

5. The Critical Role of Mesh Generation

Mesh generation remains the most significant bottleneck in the FEM workflow, often determining the success or failure of a simulation. The process involves dividing the computational domain into discrete cells that define the geometry and topology (PyLith Team, 2024).

5.1 Mesh Quality and its Impact on Numerical Accuracy

The accuracy of an FEM model is directly influenced by the refinement and quality of the mesh. In structural analysis, variables like displacements are calculated at nodes, while stresses are evaluated at Gaussian integration points. If an element is distorted, the Jacobian determinant becomes non-constant, leading to rational functions in the integral that Gaussian quadrature cannot integrate exactly (Garimella, 2025).

Key criteria for evaluating mesh quality include:

- **Aspect Ratio:** The ratio of the longest to the shortest dimension.
- **Jacobian Determinant:** A measure of the element's transformation from reference space; it must remain positive.
- **Stiffness vs. Stress Resolution:** While a coarse mesh may suffice for global stiffness, capturing stress peaks near geometric discontinuities requires localized refinement (Sagiyama *et al.*, 2024).

5.2 Tool Comparison: Gmsh vs. Cubit

Python provides the "universal glue" that orchestrates these meshing tools into a coherent chain (Dalcin *et al.*, 2011).

Table 3: Comparison of Mesh Generation Tools (Gmsh vs. Cubit)

Feature	Gmsh (PyLith Team, 2024)	Cubit (PyLith Team, 2024)
Primary Strength	Complex specification of discretization size.	Extensive suite for complex 3D geometric operations.
Cell Types	Triangular/Tetrahedral; combines them for Quads/Hex.	Native unstructured quadrilateral and hexahedral meshing.
Python API	Integrates well with external Python environments.	Requires the specific Python interpreter provided with Cubit.
Licensing	Open-source (GPL).	Commercial or Government.

A major challenge in open-source meshing workflows is the preservation of material properties and boundary condition tags. When a mesh is exported from a tool like Gmsh, metadata is often lost, requiring re-identification in the solver script a process that is highly error-prone (Dutzler *et al.*, 2021).

5.3 Adaptive Refinement and Crack Propagation

For problems involving evolving topologies, such as crack propagation, static meshes are insufficient. Continual adaptation and evolution of the mesh at each load step are essential to capture the changing geometry. Techniques like the advancing front method and Delaunay triangulation are employed to selectively refine the

region near the crack tip, often forming specialized "rosette elements" (Alshoaibin & Fageehi, 2023).

6. Scientific Computing Applications: Case Studies: The flexibility of Python-based FEM has enabled its application across a wide spectrum of scientific disciplines (Hamza, 2025).

6.1 Multiphysics in Electromagnetics and Power Electronics

Electromagnetic modeling presents unique challenges, such as the simulation of open (unbounded) regions and the need for specialized post-processing. FEniCSx has been effectively used to perform cutoff and dispersion analysis for rectangular waveguides and to treat antenna problems in a tutorial manner (Alnæs *et al.*, 2015).

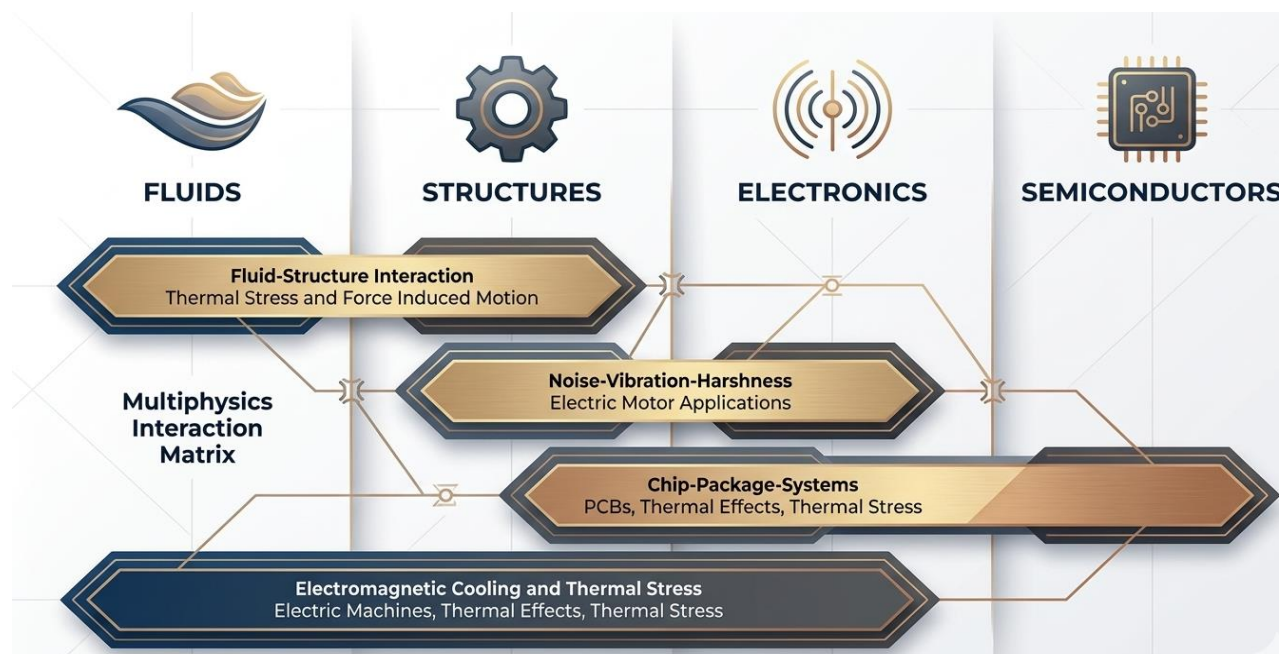


Figure1: Comprehensive Overview of Multiphysics Interactions in Engineering Systems

In the automotive industry, FEM is critical for analyzing the coupled electromagnetic-thermal performance of electric vehicle drive cycles. A 2D electromagnetic model is typically coupled with a 3D thermal model; a Python script automates the exchange of data at each operating point. The electromagnetic simulation computes losses, which are fed into the thermal model to calculate

temperature rises, adjusting material properties dynamically (JMAG Team, 2024).

6.2 Solid Mechanics and Topology Optimization

Advanced solid mechanics simulations often require specialized formulations to handle nonlinearities. For instance, simulating an infinite plate with a circular hole subjected to tension

requires the B-bar formulation to prevent "volumetric locking" in incompressible materials (Bueler, 2025).

Topology optimization has emerged as a robust design tool, allowing engineers to optimize material distribution to minimize objectives like weight while adhering to stress constraints. The FEniTop package, built on FEniCSx, integrates automatic differentiation to handle the chain rules associated with sensitivity analysis (Jia *et al.*, 2024).

6.3 Biomechanics and Fluid-Structure Interaction (FSI)

The study of biological propulsive systems, such as the metachronal motion of gossamer worm

paddles, requires complex three-dimensional FSI models. These simulations often utilize frameworks like IBAMR, which are powered by PETSc for their underlying numerics. Similarly, multiscale computational models in biomechanics integrate mechanical and biochemical stimuli to predict outcomes for skeletal muscle growth (Zilian & Habera, 2025).

7. Performance Benchmarks and Computational Efficiency

The debate regarding the performance of Python in scientific computing often centers on its interpretation overhead compared to compiled languages (Dalcin *et al.*, 2011).

Table 4: Computational Performance Comparison by Language and Task

Benchmark Task	C++ (Time)	Python/CPython (Time)	Julia (Time)
Nbody (Input 5M)	~167 ms (Garimella, 2025).	~2800 ms (Garimella, 2025).	Competitive with C++ (Garimella, 2025).
Spectral-norm (8k)	~470 ms (Garimella, 2025).	Timeout (Garimella, 2025).	Competitive with C++ (Garimella, 2025).
PDE Solver (Simple)	Baseline (Garimella, 2025).	~10-100x slower (Garimella, 2025).	Almost as fast as Fortran (Garimella, 2025).

However, these raw benchmarks can be misleading for FEM applications (Garimella, 2025). When the computationally intensive portions are offloaded to C++ or GPU kernels via libraries like DOLFINx or PyOP2, the Python overhead becomes a small fraction of the total execution time. One study showed that porting a Python solver to C++ resulted in a 100x speedup, highlighting that "branching logic in loops" is the true performance killer (Wells, 2025).

8. Emerging Frontiers: AD, PINNs, and Neuromorphic Computing

The future of FEM is increasingly intertwined with artificial intelligence and non-traditional hardware architectures (Baratta *et al.*, 2023).

8.1 Automatic Differentiation (AD) and Differentiable Physics

Automatic differentiation has transformed scientific modeling by allowing practitioners to derive implementations of finite elements automatically. Instead of manually deducing

complex formulas, compilers like ElementForge can reason from first principles to derive implementations that were previously only possible by hand (Collin, 2025).

8.2 Neuromorphic Hardware for PDE Solutions

A radical innovation is the implementation of FEM on scalable spiking neuromorphic hardware. By constructing a spiking neural network that directly implements the mathematics of the FEM, researchers have solved the Poisson equation on Intel's Loihi 2 platform. This approach achieves meaningful levels of numerical accuracy and energy-efficient scaling (Theilman & Aimone, 2026).

8.3 Physics-Informed Neural Networks (PINNs) vs. FEM

The recent success of deep neural networks has motivated the use of PINNs for numerically solving PDEs. Unlike FEM, which requires a mesh, PINNs use the PDE as a regularization term

in the loss function of a neural network (Saurav, 2025).

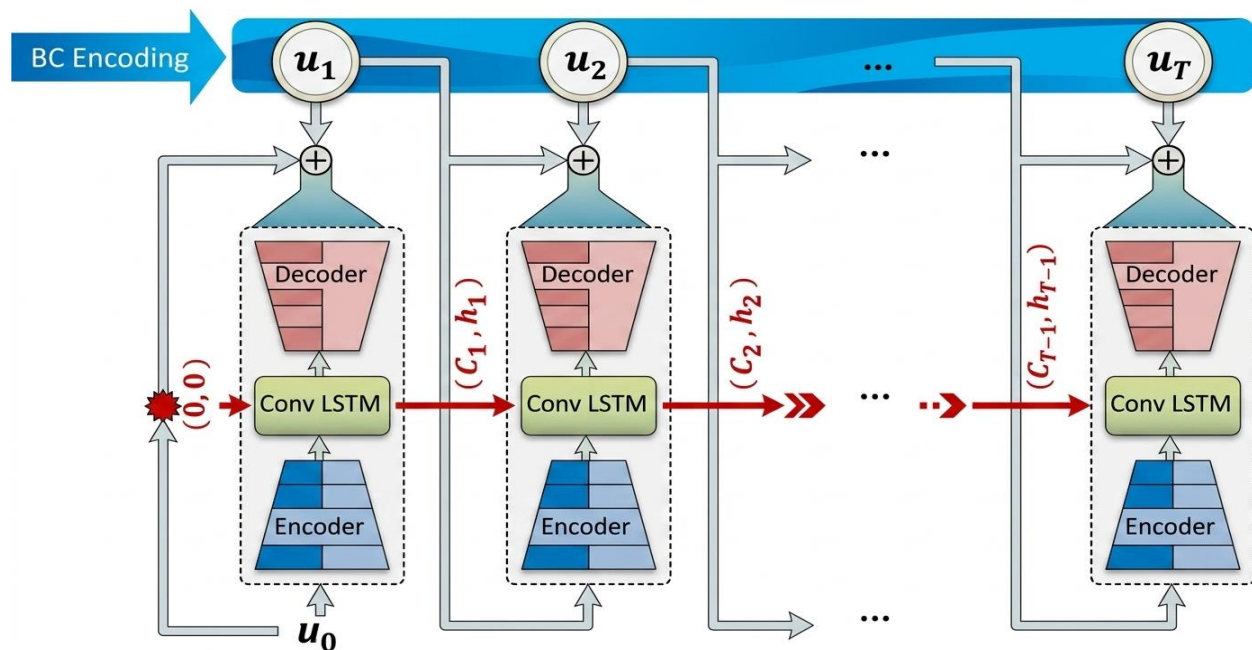


Figure 2: Architecture of a ConvLSTM-based Recurrent Encoder-Decoder Network for Sequential Data Processing

Table 5: Conceptual Comparison Between FEM and PINNs

Feature	Finite Element Method (FEM)	Physics-Informed Neural Networks (PINNs)
Discretization	Mesh-based (spatial subdivision) (Bueler, 2025).	Mesh-free (point clouds) (Saurav, 2025).
Accuracy	Generally higher and more stable (Saurav, 2025).	High for interpolation; struggles with sharp gradients (Saurav, 2025).
Cost	Efficient for forward problems (Saurav, 2025).	High training costs; fast evaluation (Saurav, 2025).
Solvers	Linear algebra/iterative solvers (Balay et al., 2026).	Optimization (Stochastic Gradient Descent) (Saurav, 2025).

9. Conclusions

The numerical solution of partial differential equations using the Finite Element Method implemented in Python has matured into a sophisticated, production-ready ecosystem that successfully bridges the gap between high-level mathematical abstraction and low-level computational efficiency. Through comprehensive analysis of modern frameworks including FEniCSx, Firedrake, and their integration with scalable backends such as PETSc and Trilinos, this paper demonstrates that

Python's role as an orchestration layer delegating computationally intensive tasks to compiled C++ or GPU kernels reduces interpretation overhead to a negligible fraction of total runtime, with true performance bottlenecks residing in algorithmic design rather than language choice. The transition to exascale computing demands matrix-free methods and GPU-aware implementations, as exemplified by MFEM, pyGinkgo, and differentiable libraries like JAX-FEM and tatva, while mesh generation persists as the primary practical bottleneck, where element quality

metrics such as Jacobian determinants and aspect ratios directly determine numerical accuracy. Emerging frontiers including automatic differentiation compilers, physics-informed neural networks offering mesh-free alternatives, and neuromorphic hardware implementations on Intel's Loihi 2 that directly embed FEM mathematics into spiking neural networks demonstrate that the field is actively evolving beyond traditional paradigms, yet the fundamental advantages of FEM's rigorous variational formulation and its ability to handle complex geometries ensure its continued dominance. Ultimately, the fusion of Python's productivity with performance-portable backends, automated verification tools like FEM-Bench, and differentiable programming for inverse design positions Python-based FEM as an indispensable methodology for exascale scientific computing and engineering discovery.

REFERENCES

- Alnæs, M. S., Logg, A., Ølgaard, K. B., Rognes, M. E., & Wells, G. N. (2014). Unified Form Language: A domain-specific language for weak formulations of partial differential equations. *ACM Transactions on Mathematical Software*, 40(2), 1–37. <https://doi.org/10.1145/2566630>
- Alnæs, M. S., Blechta, J., Hake, J., Johansson, A., Kehlet, B., Logg, A., Richardson, C., Ring, J., Rognes, M. E., & Wells, G. N. (2015). The FEniCS project version 1.5. *Archive of Numerical Software*, 3(100).
- Andrej, J., Kolev, T., & Thompson, J. L. (2024). *High performance finite elements with MFEM*. <https://doi.org/10.5194/egusphere-2025-6266>
- Balay, S., Abhyankar, S., Adams, M. F., Benson, S., Brown, J., Brune, P., Buschelman, K., Constantinescu, E. M., Dalcin, L., Dener, A., Eijkhout, V., Faibussowitsch, J., Gropp, W. D., Hapla, V., Isaac, T., Jolivet, P., Karpeev, D., Kaushik, D., Knepley, M. G., ... Zhang, J. (2026). *PETSc/TAO users manual* (ANL-21/39 - Revision 3.25). Argonne National Laboratory. <https://doi.org/10.2172/2998643>
- Bueler, E. (2025). *PETSc for partial differential equations: Numerical solutions in C and Python*. SIAM Press.
- Cimrman, R., Vackář, J., Novák, M., Čertík, O., Rohan, E., & Tůma, M. (2014). Finite element code in Python as a universal and modular tool applied to Kohn-Sham equations. *ResearchGate*.
- Collin, S. (2025). *ElementForge: A compiler for the automatic derivation of finite element implementations* (Master's thesis). Massachusetts Institute of Technology.
- Dalcin, L. D., Paz, R. R., Kler, P. A., & Cosimo, A. (2011). Parallel distributed computing using Python. *Advances in Water Resources*, 34(9), 1124–1139. <https://doi.org/10.1016/j.advwatres.2011.04.013>
- Dutzler, A., Buzzi, C., & Leitner, M. (2021). Nondimensional translational characteristics of elastomer components. *Journal of Applied Engineering Design and Simulation*, 1(1). <https://doi.org/10.24191/jaeds.v1i1.20>
- FEM-Bench. (2025). *FEM-Bench 2025: A computational mechanics benchmark for large language models*. *arXiv preprint*. <https://doi.org/10.48550/arXiv.2512.20732>
- Figueira, D. R., Becho, J. S., & Greco, M. (2025). Python implementation of the positional formulation of the finite element method for static analysis using two-dimensional elements. *Proceedings of the XLVI Ibero-Latin American Congress on Computational Methods in Engineering (CILAMCE 2025)*.
- Garimella, T. (2025, June 12). *Is Julia really that fast? I put it to test against Python's FEniCS*. Medium.
- JMAG Team. (2024). *Electromagnetic-thermal coupled drive cycle performance analysis*. TechRxiv. <https://doi.org/10.36227/techrxiv.174059928>
- NIST. (2026, February 26). *FiPy: A finite volume PDE solver in Python*. <https://www.ctcms.nist.gov/fipy/>

- PyLith Team. (2024, July 31). *PyLith user manual: Mesh generation with Cubit and Gmsh*. <https://pylith.readthedocs.io/en/v3.0.1/>
- Rathgeber, F., Ham, D. A., Mitchell, L., Lange, M., Luporini, F., McRae, A. T. T., Bercea, G. T., Markall, G. R., & Kelly, P. H. J. (2016). Firedrake: Automating the finite element method by composing abstractions. *ACM Transactions on Mathematical Software*, 43(3).
- Sagiyama, K., Mitchell, L., & Ham, D. A. (2024). *tatva: A monolithic differentiable library for large-scale simulations*. *arXiv preprint*. <https://doi.org/10.48550/arXiv.2602.12365>
- Saurav, S. S. (2025). *Can physics-informed neural networks beat finite element method?* Scribd.
- Sindhushree, P., Mandage, U. R., Pagare, I. S., & Deore, A. S. (2025). Design, fabrication and FEM analysis of multipurpose cutting machine for agricultural uses. *International Journal of Scientific Research in Engineering and Management (IJSREM)*, 9(5). <https://doi.org/10.55041/IJSREM46703>
- Theilman, B. H., & Aimone, J. B. (2026). *Solving sparse finite element problems on neuromorphic hardware*. OSTI.gov. <https://doi.org/10.2172/3006560>
- Trilinos Project. (2025). *An overview of the Trilinos project software stack*. *arXiv preprint*. <https://doi.org/10.48550/arXiv.2503.08126>
- Wells, D. (2025). *IBAMR: Immersed-Boundary adaptive mesh refinement powered by PETSc*. 2025 PETSc Annual User Meeting.
- Xue, T., Liao, S., Gan, Z., et al. (2024). JAX-FEM: A differentiable GPU-accelerated 3D finite element solver for automatic inverse design and mechanistic data science. *Computer Physics Communications*.
- Zilian, A., & Habera, M. (2025). *dolfiny: Convenience wrappers for DOLFINx and multiscale skeletal muscle growth models*. FEniCS 2025 Conference.
- Anzt, H., Chen, X., Cojean, T., Flegar, G., Grützmacher, T., Ribizel, P., & Tsai, Y. M. (2025). *pyGinkgo: A lightweight and Pythonic interface to the Ginkgo library*. *arXiv*. <https://doi.org/10.48550/arXiv.2510.08230>
- Baratta, I. A., Dean, J. P., Dokken, J. S., Habera, M., Hale, J., Richardson, C. N., Rognes, M. E., Scroggs, M. W., Sime, N., & Wells, G. N. (2023). *DOLFINx: The next generation FEniCS problem solving environment* (Version v0.7.0). Zenodo. <https://doi.org/10.5281/zenodo.10447666>
- Ham, D. A., Homolya, M., Kirby, R. C., & Mitchell, L. (2023). *An abstraction for solving multi-domain problems using finite element methods*. *arXiv*. <https://doi.org/10.48550/arXiv.2402.02523>
- Jia, Y., Wang, C., & Zhang, X. S. (2024). FEniTop, a simple FEniCSx implementation for 2D and 3D topology optimization supporting parallel computing. *Structural and Multidisciplinary Optimization*, 67(8), Article 140. <https://doi.org/10.1007/s00158-024-03818-7>
- Besson, J., Leriche, R., Foerch, R., & Cailletaud, G. (1998). Object-Oriented Programming Applied to the Finite Element Method Part II. Application to Material Behaviors. *Revue Européenne des Éléments Finis*, 7(5), 567–588. <https://doi.org/10.1080/12506559.1998.10511322>
- Maas, S. A., Ateshian, G. A., & Weiss, J. A. (2017). FEBio: History and Advances. *Annual Review of Biomedical Engineering*, 19, 279–299. <https://doi.org/10.1146/annurev-bioeng-071516-044738>
- Niiranen, J. (2021). Technical note: Open-source finite element software – from the early to the present. *Rakenteiden Mekaniikka*, 54, 172–173. <https://doi.org/10.23998/rm.113300>

- Vallisneri, M., & Babak, S. (2008). Python and XML for Agile Scientific Computing. *Computing in Science & Engineering*, 10(1), 80–87.
<https://doi.org/10.1109/mcse.2008.20>
- Ciarlet, P. G. (2002). *The finite element method for elliptic problems*. Society for Industrial and Applied Mathematics.
<https://doi.org/10.1137/1.9780898719208>
- Larson, M. G., & Bengzon, Fredrik. (2013). *The finite element method: Theory, implementation, and applications*. Springer Berlin Heidelberg.
<https://doi.org/10.1007/978-3-642-33287-6>
- Reddy, J. N. (2019). *Introduction to the finite element method* (4th ed.). McGraw-Hill Education.
- Alshoaibi, A. M., & Fageehi, Y. A. (2023). A Robust Adaptive Mesh Generation Algorithm: A Solution for Simulating 2D Crack Growth Problems. *Materials*, 16(19), 6481.
- Hamza, Z. A. (2025). Design and Simulation of Rectangular Slot Antennas Using the Finite Element Method in Python. *International journal of electrical and computer engineering systems*, 16(9), 641-650.

