Computer Applications in Engineering Education

WILEY

RESEARCH ARTICLE

A 99-Line FEM Code for Simulating a Moving Point Heat Source Over a Domain

Suparno Bhattacharyya

Department of Mechanical Engineering, Indian Institute of Technology (ISM), Dhanbad, Jharkhand, India

Correspondence: Suparno Bhattacharyya (suparno@iitism.ac.in)

Received: 8 April 2026 | **Revised:** 30 May 2026 | **Accepted:** 18 June 2026

Keywords: additive manufacturing | finite element method | moving heat source | transient heat conduction | welding simulation

ABSTRACT

Simulating transient heat transfer is crucial for computational thermal analysis in manufacturing processes such as arc welding, laser powder bed fusion, and directed energy deposition. These processes feature moving heat sources that generate steep spatial and temporal temperature gradients, which in turn strongly influence microstructure formation, residual stress development, distortion, and final mechanical properties. As a result, accurate and efficient simulation tools are invaluable for both research and industrial applications. This tutorial presents a streamlined, Python-based finite element (FE) framework for simulating three-dimensional transient heat conduction with a moving Gaussian heat source over a parallelepiped domain. Distinct from complex commercial or open-source FE codes, this approach uses the lightweight and transparent `scikit-fem` library for spatial discretization. This enables efficient assembly of system matrices and clear correspondence between mathematical formulation and code implementation. Time integration is handled using the generalized θ-method, covering Forward Euler, Backward Euler, and Crank–Nicolson schemes; the Crank–Nicolson option ($\theta = 0.5$) is chosen for its unconditional stability and second-order accuracy. Nonlinear boundary conditions due to combined convective and radiative heat losses are included via Newton–Raphson iteration with an analytically derived Jacobian. Each Newton linear system is solved using a conjugate gradient algorithm preconditioned by algebraic multigrid (AMG), ensuring efficiency for moderately fine three-dimensional meshes. The complete simulation code is concise, at roughly 99 lines, making it well-suited for teaching, self-study, or rapid prototyping. The resulting temperature fields are benchmarked against published FEniCS results for the same moving heat source scenario, exhibiting a relative field difference of only 1.5%.

1 | Introduction

Thermal management and heat transfer analysis are fundamental to a wide array of engineering applications, including electronic cooling, nuclear reactor safety, geothermal energy extraction, and advanced manufacturing. Of particular complexity are manufacturing processes involving localized and moving heat sources—such as fusion welding, laser cladding, selective laser melting (SLM), and directed energy deposition (DED). These methods pose significant challenges for thermal simulation, as a high-energy source moves through the material at a prescribed velocity, depositing energy over a small and sharply localized region. Consequently, the resulting temperature fields are highly non-uniform and evolve rapidly, producing steep spatial and temporal gradients that are challenging to capture both analytically and numerically [1, 2].

The importance of accurate thermal simulation in manufacturing extends well beyond the prediction of temperature fields. The thermal history experienced at each point in the material, which includes the sequence of peak temperatures, heating rates, and cooling rates as the heat source passes by, directly influences the material's microstructural evolution, such as grain size, phase composition, and precipitation behavior [3]. These microstructural characteristics, in turn,

dictate mechanical properties including yield strength, ductility, and fatigue life. Furthermore, the non-uniform thermal expansion and contraction produced by localized heating result in internal stresses and plastic deformation. Upon cooling, these manifest as residual stresses and part distortion [4]. Consequently, predictive simulation of these interrelated phenomena is essential for process design, optimization, and qualification in additive manufacturing and welding.

The finite element method (FEM) is widely recognized as the preferred numerical approach for thermal analysis in manufacturing, due to its capability to accommodate complex geometries, heterogeneous material properties, and general boundary conditions [5, 6]. In FEM, the spatial domain is discretized into a mesh of elements, and the unknown temperature field is approximated by piecewise polynomial functions defined on these elements. This process leads to a system of ordinary differential equations in time, or to a set of algebraic equations once time discretization is applied. For three-dimensional transient problems, the resulting linear or nonlinear algebraic systems are often large, making the use of efficient solvers and preconditioners essential for practical computations.

Several mature open-source FEM libraries are available in the Python ecosystem. Notably, FEniCS [7] and its successor Firedrake [8] offer high-level domain-specific languages to define variational forms and automate the assembly process, leveraging compiled C++ kernels through the FEniCS Form Compiler (FFC) or TSFC. SfePy [9] provides a highly modular structure that accommodates complex multi-physics simulations. Despite their powerful capabilities, these frameworks often entail a substantial installation and configuration overhead, and their abstraction layers can obscure core FEM concepts, which may render them less suitable for introductory educational purposes.

In this work, we adopt the *scikit-fem* library [10] as the primary platform for implementation. Scikit-fem intentionally employs a minimalist design: it concentrates on the core FEM assembly through intuitive NumPy-style operations, exposes the underlying mathematical structures of bilinear and linear forms via straightforward Python decorators, and permits seamless integration with any external sparse solver or preconditioner. This focus on transparency and modifiability makes scikit-fem especially advantageous for educational and exploratory code development.

This article presents a compact numerical framework for simulating three-dimensional transient heat conduction driven by a moving Gaussian heat source over a rectangular parallelepiped domain. The Gaussian beam model is widely used to approximate the energy distribution of laser sources in manufacturing simulations [11, 12], and its smooth spatial profile is well-suited for FEM discretization. The nonlinearity introduced by surface radiation boundary conditions is handled via Newton–Raphson iteration, with the consistent Jacobian derived analytically through the Fréchet derivative of the residual functional. Each Newton linear system is solved using a preconditioned conjugate gradient (PCG) method with a smoothed aggregation algebraic multigrid (AMG) preconditioner [13], a combination that offers near-optimal computational complexity for elliptic problems.

An important secondary motivation for this work is the generation of simulation data for downstream data-driven modeling. Projection-based reduced-order models, including proper orthogonal decomposition (POD)-based approaches, Gaussian process surrogates, and physics-informed neural networks (PINNs) often require resolved simulation data for basis construction, training, validation, calibration, or parameter identification [14–19]. The present Python implementation provides a transparent and modifiable data generation pipeline that can produce parametric datasets by varying process parameters, such as laser power, scan speed, and beam radius, as well as material properties or domain geometry. Furthermore, because the implementation is Python-based, the generated fields can be coupled with standard scientific machine-learning workflows based on PyTorch and JAX through array-based data exchange [20, 21].

The rest of this article is organized as follows. Section 2 presents the mathematical formulation, including the governing equation, material and domain parameters, initial and boundary conditions, and the model for the moving Gaussian heat source. Section 3 develops the weak formulation, describes the finite element spatial discretization and the generalized θ-method for time integration, and derives the Newton–Raphson solution procedure including the consistent Jacobian via the Fréchet derivative. Section 4 provides a line-by-line exposition of the Python implementation using scikit-fem, covering mesh generation, form definitions, assembly, preconditioning, and time-stepping. Section 5 presents and discusses the simulation results, including temperature field evolution, thermal gradient distributions, and solver convergence behavior. Finally, Section 6 summarizes the contributions and outlines avenues for future extension.

2 | Problem Formulation

2.1 | Governing Equation

We consider the problem of transient heat conduction in a homogeneous, isotropic solid body occupying the domain $\Omega = [0, l_x] \times [0, l_y] \times [0, l_z] \subset \mathbb{R}^3$. Under the assumption of constant material properties and no internal heat generation within the bulk, the temperature field $T = T(t, x, y, z)$ satisfies the classical heat equation:

$$\rho C_p \frac{\partial T}{\partial t} = \nabla \cdot (k \nabla T), \tag{1}$$

where ρ is the mass density (g/mm^3), C_p is the specific heat capacity (J/g·K), and k is the thermal conductivity (W/mm·K) [22]. The left-hand side represents the rate of change of thermal energy stored in a unit volume, while the right-hand side describes the net heat flux due to thermal diffusion. The ratio $\alpha = k/(\rho C_p)$ is the thermal diffusivity, which characterizes how quickly thermal disturbances propagate through the material.

Equation (1) assumes constant, temperature-independent material properties. However, in metals subjected to

high-energy laser processing, both k and C_p may vary with temperature, particularly near phase transformation temperatures. The assumption regarding constant material property is made for clarity and does not restrict the overall computational framework. If properties, such as conductivity, $k(T)$, is considered temperature-dependent, the same formulation discussed here can be extended with minimal modifications to the residual and linearization, as discussed later.

The domain dimensions and material parameters used in this study are representative of a metallic substrate (e.g., stainless steel):

$$l_x = 40\ \text{mm}, l_y = 10\ \text{mm}, l_z = 6\ \text{mm},$$

$$\rho = 8 \times 10^{-3}\ \text{g mm}^{-3}, C_p = 0.5\ \text{J g}^{-1}\ \text{K}^{-1}, k = 0.01\ \text{W mm}^{-1}\ \text{K}^{-1}. \quad (2)$$

The domain aspect ratio—long and thin in the x-direction, which is the direction of laser travel—reflects the geometry of a typical weld bead or single scan track in additive manufacturing. The thermal diffusivity computed from these parameters is $\alpha = k/(\rho C_p) = 2.5 \times 10^{-3}\ \text{mm}^2/\text{s}$, which characterizes the timescale over which thermal disturbances propagate through the domain.

2.2 | Initial and Boundary Conditions

The boundary conditions are physically motivated by the thermal environment experienced by a workpiece during laser processing: the bottom face is clamped to a substrate at ambient temperature, all lateral faces exchange heat with the surrounding environment through convection and radiation, and the top face additionally receives the incident laser energy. The six faces of the domain are labeled Γ_1 through Γ_6 as shown in Figure 1.

2.2.1 | *Dirichlet Condition (Bottom Face Γ_5)*

The bottom face of the parallelepiped ($z = 0$) is assumed to be in thermal contact with a substrate or fixture maintained at ambient temperature. This is modeled as an essential (Dirichlet) boundary condition:

$$T|_{\Gamma_5} = T_\infty = 298\ \text{K}. \quad (3)$$

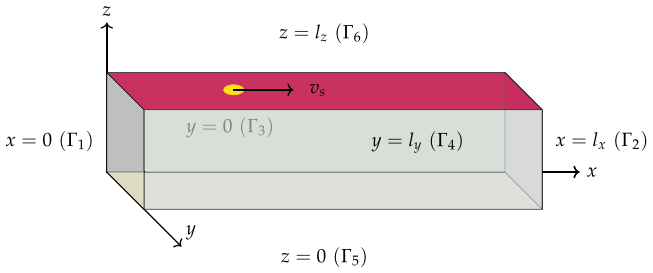

FIGURE 1 | Boundary labels for the 3D parallelepiped domain Ω. The moving heat source (yellow dot) traverses the top surface Γ_6.

In practice, this condition approximates the case of a well-cooled fixture or a thick substrate that acts as a heat sink, maintaining the bottom surface near room temperature throughout the simulation. From a mathematical standpoint, applying a Dirichlet condition on Γ_5 partitions the boundary into constrained and free parts, and the associated degrees of freedom are removed from the system via a condensation procedure [23].

2.2.2 | *Convective Heat Loss*

At all boundaries except Γ_5, the material surface loses heat to the surrounding atmosphere through convection. This is modeled by Newton's law of cooling as a Neumann-type boundary condition:

$$q_{\text{convection}} = h(T - T_\infty), h = 2 \times 10^{-5}\ \text{W/mm}^2\ \text{K}. \quad (4)$$

The convective heat transfer coefficient h is assumed constant (natural convection in air), which is a standard simplification for moderately elevated temperatures. In forced convection or gas-shielded welding scenarios, h would take larger values and may exhibit spatial variation.

2.2.3 | *Radiative Heat Loss*

At elevated temperatures, radiative heat exchange between the material surface and the environment becomes non-negligible. This is modeled by the Stefan–Boltzmann law:

$$q_{\text{radiation}} = \sigma\varepsilon\left(T^4 - T_\infty^4\right), \quad (5)$$

where $\sigma = 5.6704 \times 10^{-14}\ \text{W/mm}^2\text{K}^4$ is the Stefan–Boltzmann constant expressed in millimeter-consistent units, and $\varepsilon = 0.3$ is the total hemispherical emissivity of the surface. Note that the radiation term introduces a quartic nonlinearity in T, which is responsible for the need to solve a nonlinear system at each time step using Newton–Raphson iteration. The combined boundary heat flux on the lateral faces and the bottom-adjacent faces is thus:

$$q_{\text{surr}} = q_{\text{convection}} + q_{\text{radiation}}, \text{ on } \Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4. \quad (6)$$

2.2.4 | *Initial Condition*

Prior to the commencement of laser heating, the material is assumed to be in thermal equilibrium with its surroundings at ambient temperature. The initial condition is therefore:

$$T(0, x, y, z) = T_\infty = 298\ \text{K}, \forall\ (x, y, z) \in \Omega. \quad (7)$$

This is consistent with the Dirichlet condition on the bottom face and avoids any artificial thermal transients at the start of the simulation.

2.3 | Moving Heat Source Model

On the top surface Γ_6 (the plane $z = l_z$), the boundary heat flux includes a contribution from the laser source in addition to convection and radiation:

$$q_{\text{top}} = q_{\text{convection}} + q_{\text{laser}} + q_{\text{radiation}}, \text{ on } \Gamma_6. \tag{8}$$

The laser source is modeled as a two-dimensional Gaussian heat flux distribution with a time-dependent center position, traversing the top surface in the x-direction at constant speed [11]:

$$q_{\text{laser}} = \frac{2Q_p\eta}{\pi r^2}\exp\left(-\frac{2[(x - x_c(t))^2 + (y - y_c)^2]}{r^2}\right). \tag{9}$$

This Gaussian surface heat source model has been widely used in the welding and additive manufacturing simulation literature because it captures the essential spatial character of a focused laser or arc source, concentrated energy deposition near the beam center with exponential decay at a rate governed by the beam radius r. The prefactor $2Q_p\eta/(\pi r^2)$ ensures that, when integrated over the entire plane, the total heat input equals $Q_p\eta$, where $Q_p = 500$ W is the nominal laser power and $\eta = 0.4$ is the absorptivity (coupling efficiency). The center of the beam moves along the midline of the top surface:

$$x_c(t) = x_{\text{offset}} + v_s t, y_c = \frac{l_y}{2}, \tag{10}$$

where $x_{\text{offset}} = 4$ mm is the initial x-position and $v_s = 10$ mm/s is the scan speed. The beam radius is $r = 1.5$ mm, meaning that approximately 86% of the total power is deposited within a circle of radius r centered on the beam. The choice of $v_s = 10$ mm/s and a domain length of $l_x = 40$ mm means the beam traverses the majority of the domain in approximately 3.0 s, which—with the selected time step $\Delta t = 0.01$ s—results in about 300 time steps during active heating of any given cross-section.

The Gaussian heat source naturally couples the spatial and temporal discretizations: the beam radius $r = 1.5$ mm constrains the required mesh resolution in the x-y plane near the top surface, since the thermal gradient within the beam footprint must be adequately resolved. With the mesh resolution of $80 \times 20 \times 12$ elements over a $40 \times 10 \times 6$ mm domain, the element size near the top surface is approximately 0.5×0.5 mm, giving roughly three elements per beam radius—a moderate but acceptable resolution for capturing the Gaussian profile.

3 | Discretization and Solution Procedure

3.1 | Weak Formulation

The strong form of the problem, given by Equation (1) with the boundary and initial conditions of Section 2.2, forms a well-posed initial-boundary value problem. However, classical solutions to this problem require the temperature field to be at least twice differentiable in space, a condition that may get violated under certain circumstances such as the presence of discontinuous material properties or singular source terms. The finite element method operates instead on the *weak form*, which requires only that the solution belongs to the Sobolev space $H^1(\Omega)$—the space of square-integrable functions whose first derivatives are also square-integrable. This relaxed regularity requirement is better suited for numerical approximation and admits a broader class of admissible solutions.

To derive the weak form, we introduce the function space of admissible test functions:

$$V = \{\phi \in H^1(\Omega) : \phi|_{\Gamma_5} = 0\}, \tag{11}$$

where the homogeneous condition on Γ_5 reflects the fact that test functions must vanish where essential boundary conditions are imposed. The trial function space (for T) is the affine shift of V satisfying $T|_{\Gamma_5} = T_\infty$.

Multiplying the strong form by an arbitrary $\phi \in V$ and integrating over Ω:

$$\int_\Omega \rho C_p \frac{\partial T}{\partial t}\phi \mathrm{d}\Omega = \int_\Omega \nabla \cdot (k\nabla T)\phi \mathrm{d}\Omega. \tag{12}$$

Applying the divergence theorem to the right-hand side transfers one derivative from T to ϕ:

$$\begin{aligned}\int_\Omega \nabla \cdot (k\nabla T)\phi \mathrm{d}\Omega = &- \int_\Omega k\nabla T \cdot \nabla\phi \mathrm{d}\Omega \\ &+ \int_{\partial\Omega} k(\nabla T \cdot \mathbf{n})\phi \mathrm{d}\Gamma,\end{aligned} \tag{13}$$

where $\mathbf{n}$ is the outward unit normal. The boundary integral $\int_{\partial\Omega} k(\nabla T \cdot \mathbf{n})\phi \mathrm{d}\Gamma$ is where the natural (Neumann) boundary conditions enter: on each boundary segment, $k\partial T/\partial n$ is replaced by the prescribed heat flux (with appropriate sign convention, where outward heat flux is positive). On the Dirichlet boundary Γ_5, the test function $\phi = 0$, so that segment contributes nothing. Assembling all contributions, the weak form reads: find $T \in H^1(\Omega)$ with $T = T_\infty$ on Γ_5 such that

$$\begin{aligned}\int_\Omega \rho C_p \frac{\partial T}{\partial t}\phi \mathrm{d}\Omega + \int_\Omega k\nabla T \cdot \nabla\phi \mathrm{d}\Omega = &\int_{\Gamma_6} q_{\text{top}}\phi \mathrm{d}\Gamma \\ &+ \int_{\Gamma_1\cup\Gamma_2\cup\Gamma_3\cup\Gamma_4} q_{\text{surr}}\phi \mathrm{d}\Gamma,\end{aligned} \tag{14}$$

for all $\phi \in V$ [5, 24]. Note that the sign convention adopted here treats heat flux into the domain as positive on the right-hand side; convective and radiative terms lose heat from the domain and therefore appear with negative sign when $T > T_\infty$.

3.2 | Finite Element Approximation

The Galerkin finite element approximation replaces the infinite-dimensional function space V by a finite-dimensional subspace $V_h \subset H^1(\Omega)$, spanned by piecewise polynomial basis functions $\{\varphi_j(\mathbf{x})\}_{j=1}^N$ associated with the nodes of the mesh. The temperature field is approximated as:

$$T_h(\mathbf{x}, t) = \sum_{j=1}^{N} T_j(t)\varphi_j(\mathbf{x}), \tag{15}$$

where $T_j(t)$ are the time-dependent nodal temperatures (degrees of freedom). Substituting Equation (15) into the weak form Equation (14) and testing with each basis function φ_i yields the semi-discrete system of equations:

$$\mathsf{M}\dot{\mathbf{T}} + \mathsf{K}\mathbf{T} = \mathbf{f}(t, \mathbf{T}), \tag{16}$$

where M is the consistent mass matrix with entries $M_{ij} = \int_\Omega \rho C_p \varphi_i \varphi_j \, d\Omega$, K is the stiffness (conductivity) matrix with entries $K_{ij} = \int_\Omega k \nabla\varphi_i \cdot \nabla\varphi_j \, d\Omega$, and $\mathbf{f}$ is the right-hand side load vector incorporating all boundary flux contributions.

The domain Ω is discretized using tetrahedral elements, which tile three-dimensional geometries without introducing artificial preferred directions. Two element types are considered (see Figure 2):

- **Linear P1 elements** use the four corner nodes of each tetrahedron as degrees of freedom, with linear shape functions. The temperature gradient is therefore constant within each element, which limits accuracy when steep gradients are present. P1 elements are computationally inexpensive and easy to implement.
- **Quadratic P2 elements** augment the four corner nodes with six midpoint nodes on the edges, for a total of ten nodes per element. The shape functions are quadratic, enabling a much better representation of curved temperature profiles. For smooth problems, P2 elements exhibit $O(h^3)$ convergence in the L^2 norm versus $O(h^2)$ for P1, where h is the mesh size [25].

The computational cost of P2 elements is higher per element due to more degrees of freedom, but the improved accuracy-per-degree-of-freedom trade-off generally favors P2 for problems with smooth but rapidly varying fields, such as those driven by Gaussian heat sources.

3.3 | Time Integration Using the Generalized θ-Method

To complete the discretization, the semi-discrete system Equation (16) must be integrated in time. We partition the time interval $[0, t_f]$ into N_t uniform steps of size Δt, with $t_n = n\Delta t$ for $n = 0, 1, \ldots, N_t$. The generalized θ-method approximates the time derivative by a finite difference and evaluates the spatial operators at a θ-weighted combination of the current and previous time levels:

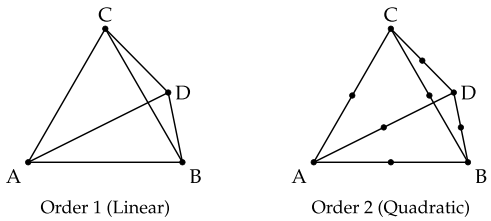

FIGURE 2 | Schematic representation of the tetrahedral finite elements used in the simulations: first-order linear element with corner nodes only (left) and second-order quadratic element with additional midside nodes (right).

$$\mathsf{M}\frac{\mathbf{T}^{n+1} - \mathbf{T}^n}{\Delta t} + \mathsf{K}[\theta\mathbf{T}^{n+1} + (1-\theta)\mathbf{T}^n] = \theta\mathbf{f}^{n+1} + (1-\theta)\mathbf{f}^n, \tag{17a}$$

$$\Rightarrow \left(\frac{1}{\Delta t}\mathsf{M} + \theta\mathsf{K}\right)\mathbf{T}^{n+1} - \left(\frac{1}{\Delta t}\mathsf{M} - (1-\theta)\mathsf{K}\right)\mathbf{T}^n = \theta\mathbf{f}^{n+1} + (1-\theta)\mathbf{f}^n. \tag{17b}$$

where $\theta \in [0, 1]$ is the implicitness parameter. The three canonical choices are as follows:

- $\theta = 0$: **Forward Euler** (fully explicit). The right-hand side is evaluated entirely at the known time level t_n, so no system of equations needs to be solved. However, the method is only first-order accurate in time and subject to a stability constraint on Δt: for the heat equation, the Courant–Friedrichs–Lewy (CFL) condition requires $\Delta t \lesssim h^2/(2\alpha)$, which becomes prohibitively small for fine meshes.
- $\theta = 1$: **Backward Euler** (fully implicit). A linear system must be solved at each step, but the method is unconditionally stable and allows arbitrarily large time steps. It is first-order accurate in time and introduces numerical dissipation (artificial damping), which can smooth out short-time-scale features.
- $\theta = 0.5$: **Crank–Nicolson**. The trapezoidal averaging of the spatial operators at t_n and t_{n+1} gives second-order accuracy in time while preserving unconditional stability. Formal proofs of these properties are provided in the appendix. The Crank–Nicolson scheme has been used in several transient thermal and thermo-elastic analyses involving temperature evolution in structural and continuum systems [26–30]. It is useful when temporal accuracy is important and the time step is not too large, since its limited numerical dissipation helps preserve physical oscillations in the computed response.

In the present work, $\theta = 0.5$ (Crank–Nicolson) is employed. With $\Delta t = 0.01$ s and a beam radius of 1.5 mm traveling at 10 mm/s, the beam center moves $v_s \Delta t = 0.1$ mm per time step—one-fifth of a full element width—ensuring smooth temporal resolution of the beam position.

3.4 | Nonlinear Residual and Newton–Raphson Linearization

Due to the quartic nonlinearity in the radiative boundary condition, Equation (17a) is a nonlinear system in $\mathbf{T}^{n+1}$ at each time step. Rearranging the θ-method equation, the residual vector is defined as

$$\mathbf{R}(\mathbf{T}^{n+1}) = \left(\frac{\mathsf{M}}{\Delta t} + \theta\mathsf{K}\right)\mathbf{T}^{n+1} - \left(\frac{\mathsf{M}}{\Delta t} - (1-\theta)\mathsf{K}\right)\mathbf{T}^n - \theta\mathbf{f}(\mathbf{T}^{n+1}) - (1-\theta)\mathbf{f}(\mathbf{T}^n) \tag{18}$$

Newton–Raphson iteration solves this system by linearizing $\mathbf{R}$ about the current iterate $\mathbf{T}_k^{n+1}$. Starting from the scalar analogy (see Figure 3), where the method finds the root of $R(T) = 0$ by

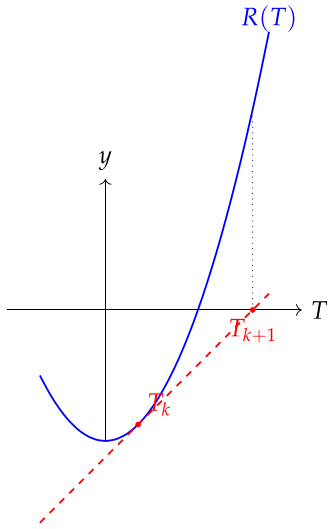

FIGURE 3 | Schematic illustration of Newton's method for finding the root of the nonlinear residual equation $R(T) = 0$ in one dimension. The tangent to $R(T)$ at T_k intersects the T-axis at the updated estimate T_{k+1}.

repeatedly constructing the tangent line at T_k and intersecting it with the T-axis:

$$T_{k+1} = T_k - \frac{R(T_k)}{R'(T_k)}, \tag{19}$$

the vector generalization yields the Newton update:

$$\mathbf{T}_{k+1}^{n+1} = \mathbf{T}_k^{n+1} - \left[\mathsf{J}\left(\mathbf{T}_k^{n+1}\right)\right]^{-1}\mathbf{R}\left(\mathbf{T}_k^{n+1}\right), \tag{20}$$

where $\mathsf{J}(\mathbf{T}) = \partial\mathbf{R}/\partial\mathbf{T}$ is the Jacobian matrix. In practice, we avoid forming the matrix inverse and instead solve the linear system:

$$\mathsf{J}\left(\mathbf{T}_k^{n+1}\right)\delta\mathbf{T} = -\mathbf{R}\left(\mathbf{T}_k^{n+1}\right), \tag{21}$$

and update $\mathbf{T}_{k+1}^{n+1} = \mathbf{T}_k^{n+1} + \delta\mathbf{T}$. This is iterated until the norm of the correction $\|\delta\mathbf{T}\|$ falls below a prescribed tolerance. The use of the *consistent* Jacobian (derived exactly from the residual, rather than approximated) ensures quadratic convergence near the solution, meaning the number of significant digits in the error approximately doubles at each iteration. This is confirmed in practice by the observation of 2–3 Newton iterations per time step even for tight tolerances of 10^{-11}.

3.5 | Determination of the Jacobian Via the Fréchet Derivative

The Jacobian is computed analytically by taking the Fréchet (directional) derivative of the residual functional. The Fréchet derivative of $R(T_h^{n+1})$ in the direction δT is defined as the limit:

$$R'\left(T_h^{n+1}\right)[\delta T] = \frac{\mathrm{d}}{\mathrm{d}\omega}R\left(T_h^{n+1} + \omega\delta T\right)\Big|_{\omega=0}. \tag{22}$$

This measures how the residual changes when the temperature field is perturbed by a small increment $\omega\delta T$. Each term in the residual is differentiated separately:

3.5.1 | *Time-Derivative (Mass) Term*

The contribution to the residual from the mass term is linear in T_h^{n+1}, so its derivative is simply:

$$\frac{\mathrm{d}}{\mathrm{d}\omega}\int_\Omega \rho C_p \frac{\left(T_h^{n+1} + \omega\delta T\right) - T_h^n}{\Delta t}\phi\,\mathrm{d}\Omega\Big|_{\omega=0} = \int_\Omega \rho C_p \frac{\delta T}{\Delta t}\phi\,\mathrm{d}\Omega. \tag{23}$$

This gives the consistent mass matrix term $\mathsf{M}/\Delta t$ in the Jacobian.

3.5.2 | *Diffusion Term*

The diffusion term at the new time level is also linear in T_h^{n+1}, contributing:

$$\frac{\mathrm{d}}{\mathrm{d}\omega}\left[\theta\int_\Omega k\nabla\left(T_h^{n+1} + \omega\delta T\right)\cdot\nabla\phi\,\mathrm{d}\Omega\right]_{\omega=0} = \theta\int_\Omega k\nabla(\delta T)\cdot\nabla\phi\,\mathrm{d}\Omega. \tag{24}$$

The term at the old time level T_h^n is independent of T_h^{n+1} and therefore contributes zero to the Jacobian.

3.5.3 | *Nonlinear Boundary Flux Term*

This is where the nontrivial contribution arises. The boundary heat flux $q(T) = h(T - T_\infty) + \sigma\varepsilon(T^4 - T_\infty^4)$ depends nonlinearly on T. Its derivative with respect to T is:

$$q'(T) = h + 4\sigma\varepsilon T^3, \tag{25}$$

which is the sum of the linearized convection coefficient and the linearized radiation coefficient. The contribution to the Jacobian from the boundary terms is:

$$-\theta\int_{\Gamma_{1,2,3,4,6}}\left[h + 4\sigma\varepsilon\left(T_h^{n+1}\right)^3\right]\phi\delta T\,\mathrm{d}\Gamma. \tag{26}$$

The factor $4T^3$, at temperatures significantly above ambient, grows rapidly and dominates the Jacobian modification relative to the linear convection coefficient.

3.5.4 | *Assembled Jacobian*

Collecting all contributions and substituting $\delta T = \sum_{k=1}^{N}\delta T_k\varphi_k(\mathbf{x})$, the complete Jacobian matrix has entries:

$$J_{ij} = \int_\Omega \frac{\rho C_p}{\Delta t}\varphi_i\varphi_j \mathrm{d}\Omega + \theta\int_\Omega k\nabla\varphi_i \cdot \nabla\varphi_j \mathrm{d}\Omega - \theta\int_{\Gamma_{1,2,3,4,6}} \left[h + 4\sigma\varepsilon\left(T_h^{n+1}\right)^3\right]\varphi_i\varphi_j \mathrm{d}\Gamma. \tag{27}$$

The first two terms depend only on fixed material parameters and can be precomputed (they do not change between Newton iterations at the same time step). The third term depends on the current iterate T_h^{n+1} and must be reassembled at each Newton step, contributing the nonlinear temperature-dependent stiffness modification due to radiation.

4 | Implementation in scikit-fem

The Python library `scikit-fem` is designed with the philosophy that finite element assembly should be transparent, composable, and as close to the mathematical notation as possible [10]. Unlike heavier frameworks that compile variational forms to optimized C++ at runtime, scikit-fem operates as a pure Python/NumPy library, trading some computational speed for complete transparency and zero-overhead installation. Its core abstraction is the *form*: a bilinear form `@BilinearForm` defines the integrand of a matrix assembly, and a linear form `@LinearForm` defines a vector assembly. These map one-to-one onto the mathematical weak form integrals, making the code a nearly executable version of the equations in Section 3.

4.1 | Imports and Global Parameters

The simulation requires four primary Python packages:

- `scikit-fem`: mesh generation, basis construction, and finite element assembly.
- `pyamg` [13]: algebraic multigrid preconditioning for the Newton linear systems.
- `numpy` [31]: all array operations and mathematical functions.
- `scipy` [32]: sparse matrix storage formats and iterative linear solvers (PCG).

Version pinning for `numpy` ($\lesssim 1.26$) and `scipy` ($\lesssim 1.13$) is required due to internal scikit-fem dependencies on legacy sparse matrix interfaces. These can be installed via:

```
pip install scikit-fem pyamg "numpy==1.26" "scipy==1.13"
```

```
import numpy as np
from pyamg import smoothed_aggregation_solver
from skfem import *
from skfem.helpers import grad, dot

lx, ly, lz = 40.0, 10.0, 6.0
nx, ny, nz = 2 * int(lx), 2 * int(ly), 2 * int(lz)
rho, cp, k = 8e-3, 0.5, 0.01
T_infty, h = 298.0, 2e-5
Rboltz, emiss = 5.6704e-14, 0.3
scan_speed, r, Qp, eta = 10.0, 1.5, 500.0, 0.4
dt, theta, t_end = 0.01, 0.5, 3.0
ts = np.arange(0.0, t_end + dt, dt)
save_every = 1
```

Listing 1: Imports and global parameters.

Parameter notes:

- The mesh resolution `nx, ny, nz = 80, 20, 12` gives element sizes of approximately $0.5 \times 0.5 \times 0.5$ mm, which is one-third of the beam radius $r = 1.5$ mm—sufficient to resolve the Gaussian profile.
- The value `scan_speed = 10` corresponds to $v_s = 10$ mm/s.
- Setting `theta = 0.5` corresponds to the Crank–Nicolson scheme. Changing this to `1.0` switches to the more dissipative but unconditionally stable Backward Euler scheme, which may be preferable for stiffer problems.
- `t_end = 300 * 0.01 = 3.0` s corresponds to the total simulation duration, during which the beam center travels from $x_c(0) = 4$ mm to $x_c(3) = 34$ mm.
- `save_every` indicates how frequently the simulation data is saved.

4.2 | Mesh Generation and Boundary Markers

Scikit-fem generates structured tetrahedral meshes from tensor products of one-dimensional grids using `MeshTet.init_tensor`. This creates a regular hexahedral lattice, then subdivides each hexahedron into five tetrahedra, yielding a conforming tetrahedral mesh with uniform refinement. Named boundary labels are attached using the `with_boundaries` method, which accepts a dictionary of Boolean-valued lambda functions that identify boundary facets by their coordinate values.

```
        # Mesh and boundaries
        mesh = MeshTet.init_tensor(
            np.linspace(0, lx, nx + 1),
            np.linspace(0, ly, ny + 1),
            np.linspace(0, lz, nz + 1),
        ).with_boundaries({
            "left": lambda x: np.isclose(x[0], 0.0),
            "right": lambda x: np.isclose(x[0], lx),
            "front": lambda x: np.isclose(x[1], 0.0),
            "back": lambda x: np.isclose(x[1], ly),
            "bottom": lambda x: np.isclose(x[2], 0.0),
            "top": lambda x: np.isclose(x[2], lz),
        })

        element = ElementTetP1() #For higher accuracy convert to ELementP2()
        basis = Basis(mesh, element)
        bottom_dofs = basis.get_dofs("bottom").all()
        facets = np.unique(np.concatenate([mesh.boundaries[n] for n in ("left",
             "right", "front", "back", "top")]))
        fb = FacetBasis(mesh, element, facets=facets)
        ft = FacetBasis(mesh, element, facets=mesh.boundaries["top"])
        laser_t = {"time": 0.0}
```

Listing 2: Mesh generation and boundary identification.

Notes on mesh and basis construction:

- `np.isclose` is used rather than exact equality to handle floating-point rounding in the node coordinates; this is essential for reliable boundary identification.
- The `Basis` object is the central data structure in scikit-fem. It encodes the mapping between reference and physical elements, precomputes quadrature points and weights, and provides interpolation of nodal fields at quadrature points. All assembly operations are performed via the `Basis` object.
- `basis.N` returns the total number of degrees of freedom, which includes both corner and mid-edge nodes. On an $80 \times 20 \times 12$ tetrahedral mesh with P1 elements, `basis.N` is approximately 22,113, reflecting the DOF count.
- `bottom_dofs` is a DOF index set used to enforce the Dirichlet condition by condensation (elimination of those rows and columns from the assembled system).
- `laser` defines the moving Gaussian surface heat flux on the top boundary. Since it is assembled on `ft`, it acts as a boundary flux rather than a volumetric source.

4.3 | Weak Forms for the Governing Equations

Each term in the weak form Equation (14) and Jacobian Equation (27) is defined as a separate scikit-fem form object. The scikit-fem decorator `@LinearForm` wraps a function `f(v, p)` that returns the integrand of a vector assembly, where `v` is the test function value and `p` is a parameter dictionary holding interpolated field values. Similarly, `@BilinearForm` wraps `f(u, v, p)` where `u` is the trial function. The helper functions `grad()` and `dot()` from `skfem.helpers` operate element-wise and broadcast correctly over all quadrature points.

```
        @BilinearForm
        def transient_term(u, v, w): return u * v

        @BilinearForm
        def diffusion_term(u, v, w): return dot(grad(u), grad(v))

        @LinearForm
        def qloss(v, p):
            T = (1.0 -theta) * p["prev"] + theta * p["trial"]
            return (h * (T -T_infty) + Rboltz * emiss * (T**4 -T_infty**4)) * v

        @BilinearForm
        def jloss(u, v, p):
            T = (1.0 -theta) * p["prev"] + theta * p["trial"]
            return theta * (h + 4.0 * Rboltz * emiss * T**3) * u * v

        @LinearForm
        def laser(v, w):
            xc, yc = 0.1 * lx + scan_speed * laser_t["time"], ly / 2.0
            rsq = (w.x[0] -xc) ** 2 + (w.x[1] -yc) ** 2
            return (2.0 * Qp * eta / (np.pi * r**2)) * np.exp(-2.0 * rsq / r
                **2) * v
```

Listing 3: Weak form definitions.

Design notes:

- `transient_term` defines the mass bilinear form and yields the matrix matrix.
- `diffusion_term` defines the standard diffusion bilinear form and yields the stiffness matrix K. The θ-weighting is applied later through `A0` and `B0`, not inside this form.
- `qloss` defines the nonlinear boundary heat-loss term due to convection and radiation, evaluated on the selected boundary facets using the θ-interpolated temperature based on the previous solution and the current Newton iterate.
- `jloss` defines the boundary Jacobian associated with `qloss`. Since it is assembled on `fb`, it contributes only on the exposed boundary facets.
- The domain contributions are formed algebraically through `A0` and `B0`, while the nonlinear boundary contributions are added through `qloss` and `jloss`.

4.4 | Assembly of Residual and Jacobian

The function `assemble` encapsulates the assembly procedure at each Newton iteration. It accepts the current iterate `u` and the previous time-step solution `u_old`, updates the laser position at the intermediate time `t_theta`, interpolates both fields onto the boundary facet basis, and assembles the residual and Jacobian contributions. The domain terms are formed algebraically through `A0` and `B0`, (defined below) while the nonlinear boundary loss and its Jacobian are assembled through `qloss` and `jloss`. The laser contribution is assembled separately on the top boundary through `laser`.

```
        M = asm(mass, basis).tocsr()
        K = asm(stiff, basis).tocsr()
        mfac = rho * cp / dt
        A0 = mfac * M + theta * k * K
        B0 = ((1.0 -theta) * k * K -mfac * M).tocsr()

        def assemble(u, u_old, t_theta):
            laser_t["time"] = t_theta
            ub, uob = fb.interpolate(u), fb.interpolate(u_old)
            r = A0 @ u + B0 @ u_old
            r += asm(qloss, fb, trial=ub, prev=uob)
            r -= asm(laser, ft)
            J = A0 + asm(jloss, fb, trial=ub, prev=uob)
            return J, r
```

Listing 4: Residual and Jacobian assembly.

Assembly notes:

- `M = asm(mass, basis)` assembles the global mass matrix M, and `K = asm(stiff, basis)` assembles the global diffusion matrix K.
- `mfac = rho * cp/dt` defines the scaling factor $\rho C_p/\Delta t$ associated with the transient term.
- `A0 = mfac * M + theta * k * K` collects the contributions that multiply the current unknown **u** in the residual, namely the transient term and the implicit part of the diffusion term.
- `B0 = ((1.0 - theta) * k * K - mfac * M)` collects the contributions involving the previous time-step solution $\mathbf{u}_{\text{old}}$, namely the explicit part of the diffusion term and the known transient contribution.
- In `assemble(u, u_old, t_theta)`, the laser time is first updated to the intermediate time level `t_theta`. The fields `ub` and `uob` are then obtained by interpolating the current Newton iterate and the previous time-step solution onto the boundary facet basis.
- `r = A0 @ u + B0 @ u_old` forms the domain part of the residual. The nonlinear boundary heat-loss contribution is then added through `asm(qloss,)`, and the applied laser heat flux is subtracted through `asm(laser, ft)`.
- `J = A0 + asm(jloss,)` forms the Jacobian matrix. Thus, the domain Jacobian is given by `A0`, while the nonlinear boundary correction is supplied by `jloss`.
- The function returns the pair `(J, r)`, that is, the Jacobian and residual required by the Newton iteration.

4.5 | Preconditioning and Newton's Method

Each Newton linear system $J\delta\mathbf{T} = -\mathbf{R}$ is a large, sparse, symmetric positive definite system (after condensation of the Dirichlet DOFs). The system size for the present mesh is on the order of 10^5 unknowns, which makes direct factorization methods (LU, Cholesky) impractical due to memory requirements for the fill-in of the sparse factors. Instead, an iterative Krylov method is employed.

The **Preconditioned Conjugate Gradient (PCG)** method is the iterative solver of choice for symmetric positive definite systems. Starting from an initial guess, PCG generates a sequence of iterates that minimize the energy norm of the error over a growing Krylov subspace. The convergence rate of PCG is governed by the condition number $\kappa(J)$ of the Jacobian: well-conditioned systems converge in few iterations, while poorly conditioned systems require many. For FEM matrices arising from elliptic PDEs, the condition number scales as $\kappa \sim O(h^{-2})$, which grows rapidly as the mesh is refined.

To restore mesh-independent convergence, a **Smoothed Aggregation Algebraic Multigrid (SA-AMG)** preconditioner [13] is applied. AMG constructs a hierarchy of progressively coarser problems by aggregating degrees of freedom, then uses a V-cycle (representing a down-and-up pass through this hierarchy) of pre/post-smoothing (damped Jacobi or Gauss–Seidel) and coarse-grid correction to efficiently annihilate error components at all frequencies. Here, smoothing refers to inexpensive iterations that quickly remove short-scale oscillations in the current solution error. SA-AMG is particularly effective for scalar elliptic problems and achieves near-optimal O(N) complexity, meaning the total solve time scales linearly with the number of unknowns.

```
        # Newton solver
        def newton(u, u_old, t_theta, ml=None, tol=1e-8, maxit=20):
            u = u.copy()
            u[bottom_dofs] = T_infty
            for _ in range(maxit):
                J, r = assemble(u, u_old, t_theta)
                A, b, x, I = condense(J, -r, D=bottom_dofs)
                A = A.tocsr()
                ml = smoothed_aggregation_solver(A) if ml is None else ml
                if hasattr(ml, "change_solve_matrix"): ml.change_solve_matrix(A)
                du_red = solve(A, b, solver=solver_iter_pcg(
                    M=ml.aspreconditioner(cycle="V"), rtol=1e-6, atol=1e-8,
                        verbose=False))
                du = x.copy()
                du[I] = du_red
                u_new = u + du
                u_new[bottom_dofs] = T_infty
                if np.linalg.norm(u_new -u) < tol: return u_new, ml
                u = u_new
            raise RuntimeError("Newton solver did not converge.")
```

Listing 5: Preconditioner and Newton solver.

Design notes:

- `newton` solves the nonlinear system at a given time level by applying Newton iterations to the current unknown vector `u`, using `u_old` as the known solution from the previous time step.
- The initial guess is copied from the input field, and the Dirichlet condition on the bottom boundary is enforced immediately by setting `u[bottom_dofs] = T_infty`.
- At each iteration, `assemble(u, u_old, t_theta)` returns the current Jacobian matrix and residual vector corresponding to the present Newton iterate.
- `condense(J, -r, D=bottom_dofs)` eliminates the prescribed degrees of freedom from the linearized system and forms the reduced system to be solved for the Newton correction.
- The reduced matrix is solved with a preconditioned conjugate gradient method. The algebraic multigrid hierarchy `ml` is built on the first iteration and then reused, with the system matrix updated when supported by the solver object. This is efficient, but if the Jacobian changes a lot, preconditioner quality can degrade.
- The reduced correction `du_red` is lifted back to the full space through `du`, and the temperature field is updated as `u_new = u + du`.
- After each update, the bottom-boundary temperature is enforced again to maintain the essential boundary condition exactly.
- Convergence is checked through the norm of the change between successive iterates, `np.linalg.norm(u_new - u)`. If this quantity falls below `tol`, the function returns the converged solution together with the multigrid object for reuse in later calls.
- If convergence is not reached within `maxit` iterations, the function raises an error indicating failure of the Newton solve.

4.6 | Time-Stepping Loop

The outermost loop advances the simulation from $t = 0$ to $t = t_{\text{end}}$ by iterating over the pre-computed time array `ts`. At each step, the Crank–Nicolson effective time t_θ is computed, the laser center is updated, and Newton's method is called to converge the nonlinear system.

Time-stepping notes:

- The initial condition `u_old = np.full(basis.N, T_infty)` sets all nodal temperatures to 298 K, consistent with the initial condition in Section 2.2. For P2 elements, this includes both corner nodes and mid-edge nodes.
- The expression $t_\theta = (1-\theta)t_n + \theta t_{n+1}$ evaluates the effective simulation time at which the laser position is computed. For $\theta = 0.5$, this gives $t_\theta = (t_n + t_{n+1})/2$, the midpoint time, which ensures the laser position is evaluated at the center of the time interval.
- `u_old.copy()` is passed to `newton_solver` as the initial guess for $\mathbf{T}_k^{n+1}$. Using the previous time step solution as the initial guess is typically excellent for small Δt, as the temperature change between steps is small. This warm-starting strategy is key to achieving convergence in 2–3 Newton iterations.
- The converged solution at each time step is appended to `u_sol`, building up the full spatio-temporal solution dataset. This array can be exported in `.npz`, `.vtk`, or HDF5 format for visualization (e.g., in ParaView) or downstream data-driven modeling.

5 | Results and Discussion

Figure 4 presents a top view and a cross-sectional view of a representative temperature snapshot visualized in ParaView. The moving Gaussian source produces an asymmetric thermal field: a compact high-temperature region remains attached to the current laser position, while a trailing wake extends behind it along the scan direction. Material ahead of the source is still near ambient, whereas material behind it retains heat and cools over a finite length through conduction, convection, and radiation. This front–rear imbalance gives the thermal footprint and the heat-affected zone (HAZ) a tear-drop or comet-like form.

The cross-sectional view shows that the peak temperature occurs near the top surface under the beam center, after which heat penetrates into the body. The thermal gradient,

$$\nabla T_h(\mathbf{x}) = \sum_{j=1}^{N} T_j \nabla \varphi_j(\mathbf{x}), \tag{28}$$

```
        # Time-stepping loop
        u_old = np.full(basis.N, T_infty)
        u_sol, ml = [u_old.copy()], None
        for n in range(len(ts) -1):
            t_theta = (1.0 -theta) * ts[n] + theta * ts[n + 1]
            u_old, ml = newton(u_old, u_old, t_theta, ml=ml)
            if (n + 1)%
            print(f"step={n+1:03d}, time={ts[n+1]:.4f}, Tmax={u_old.max():.6f}"
                )
```

Listing 6: Time-stepping loop.

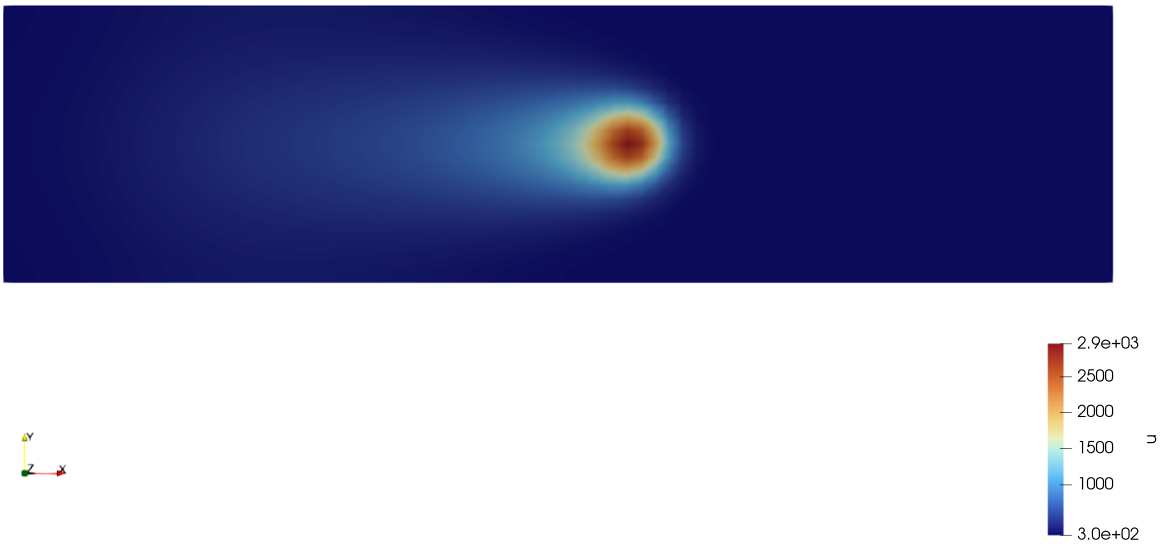

FIGURE 4 | Temperature field obtained from the transient heat-transfer simulation and visualized in ParaView: (a) top view of the computational domain and (b) longitudinal cut section through the domain. The localized high-temperature region indicates the position of the moving heat source and the surrounding thermal diffusion zone.

is therefore strongest near the source and in the through-thickness direction. In the scan direction, the gradient field is also asymmetric: the leading edge is steep because the laser approaches colder material, while the trailing edge decays over a longer distance after the source has passed. The resulting HAZ is short in front of the beam and elongated behind it.

As the simulation proceeds, the location of the temperature peak follows the laser path, while the thermal wake records the recent heating history. The field therefore contains both the instantaneous source location and the cumulative effect of prior heating, which is relevant in processes where repeated thermal exposure influences subsequent response.

In Figure 5, we assess the accuracy of scikit-fem results against a published FEniCS benchmark problem in [33], which has the same problem setup and the same structured mesh used to discretize the parallelepiped. The simulations were carried out in a 12th Gen Intel(R) Core(TM) i7-12700K CPU (32 GB RAM) using a single core. A direct point-by-point comparison of the two temperature fields is also presented in Figure 6, without any remapping of the temperature values onto the mesh. A visual inspection suggests that the two temperature distributions are nearly identical. We observed, however, that the FEniCS solution was at least 9 × slower, while the relative difference remained only 1.5%. The maximum temperature difference between the two simulations was about 78 K, with peak temperatures of 2960 K and 3038 K, respectively.

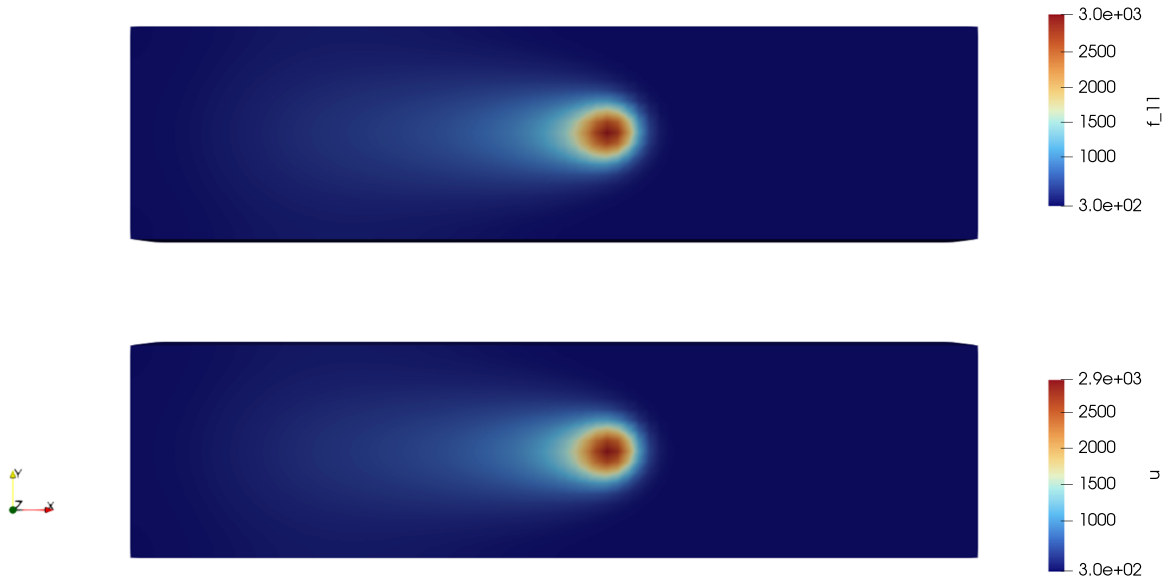

FIGURE 5 | Simulation results verified against FEniCS [33]. The 3D visualization shows the temperature field at $t = 180$ in a parallelepiped domain of dimensions $40 \times 10 \times 6$, discretized using P1 tetrahedral elements on an $80 \times 20 \times 12$ mesh. A direct comparison between the scikit-fem (bottom) and FEniCS solutions (top) shows close agreement, with a relative difference of 1.5% and a maximum temperature difference of about 78 K. The peak temperatures obtained from the two simulations were 2960 K and 3038 K, respectively. On the same machine with 32 GB RAM, the FEniCS simulation took 390 s, whereas the scikit-fem simulation took 42 s.

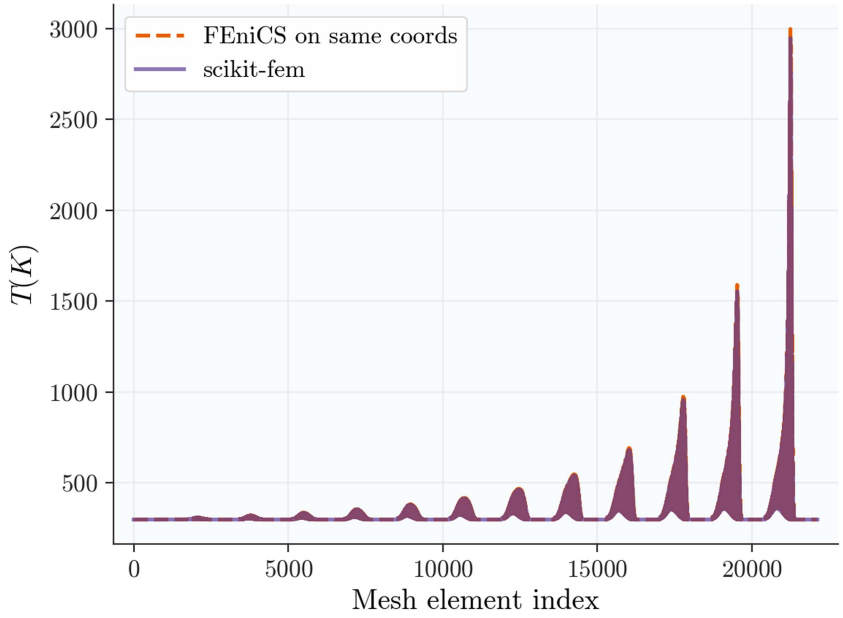

FIGURE 6 | Direct point-by-point comparison of the temperature distribution obtained via fenics and scikit-fem against mesh element index as presented in Figure 5.

Temperature distribution due to several other laser trajectories have been shown in Figure 7.

We further observed that the speed-up became more pronounced with mesh refinement. For a mesh containing $200 \times 50 \times 30$ elements over the same parallelepiped, scikit-fem required ~1.6 s per time step, whereas FEniCS required about ~210 s per time step. This advantage may not persist indefinitely, however, as FEniCS is expected to become more competitive for much larger problems through stronger solver infrastructure and parallel scalability.

To further quantify computational cost, Tables 1 and 2 report two timing studies for the present scikit-fem implementation. Table 1 shows the effect of mesh resolution for a fixed number of time steps. As the number of DOFs increases from 3157 to 317781, the average time per step increases from 0.029 s to 2.246 s, showing an approximately proportional increase with problem size. Table 2 shows the effect of simulation duration for a fixed $80 \times 20 \times 12$

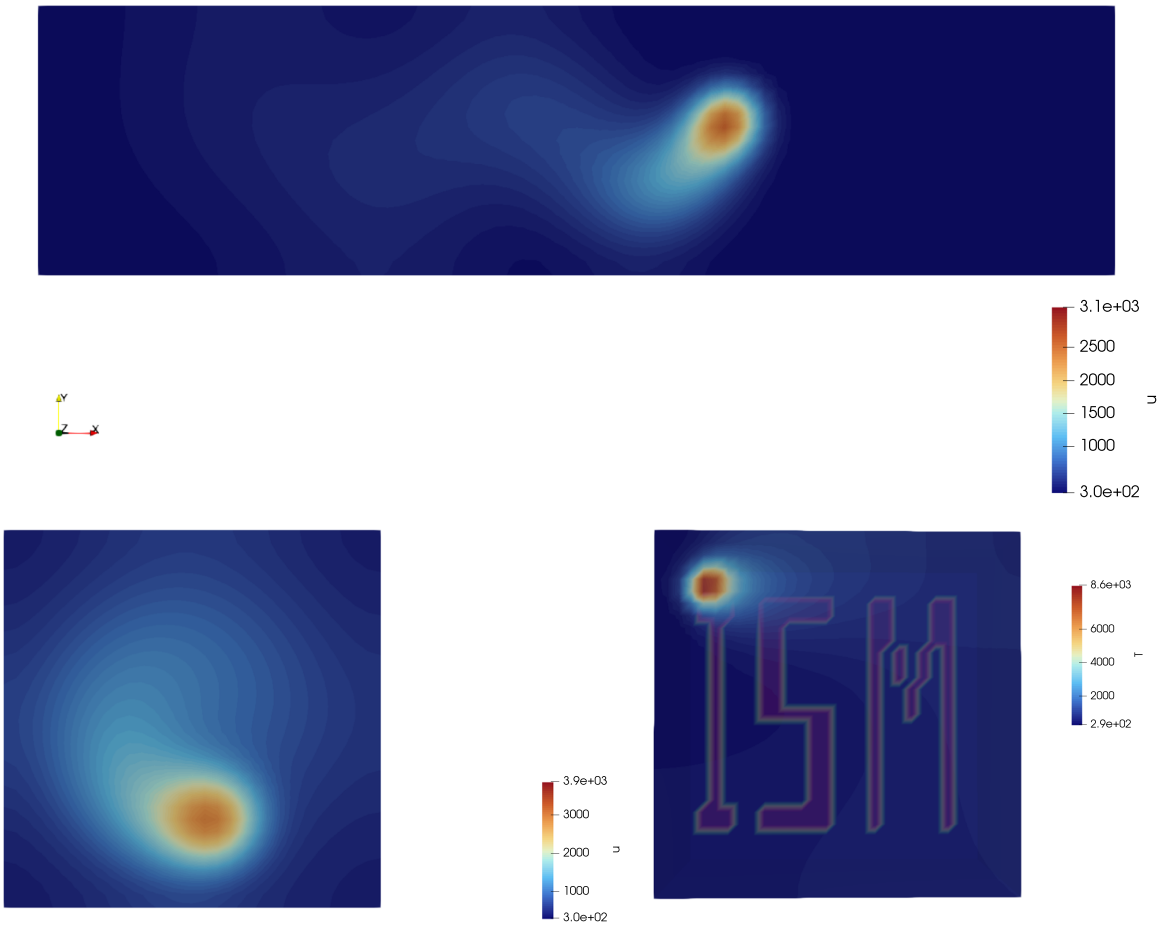

FIGURE 7 | Simulation of the temperature distribution due to various laser trajectories performed using the code: sinusoidal (top), sprial (bottom left), and rectilinear (bottom right).

TABLE 1 | Effect of mesh resolution on computational cost.

Mesh size	Elements (P1)	DOFs	Steps	Avg. Newton iters./step	Avg. time/step (s)	Total time (s)
40 × 10 × 6	14400	3157	100	3.00	0.029	2.99
80 × 20 × 12	115200	22113	100	3.00	0.140	14.88
120 × 30 × 18	388800	71269	100	3.00	0.422	45.13
200 × 50 × 30	1800000	317781	100	3.00	2.246	239.71

TABLE 2 | Effect of simulation duration on computational cost.

Mesh size	Elements (P1)	DOFs	Steps	Sim. time (s)	Avg. Newton iters./step	Avg. time/step (s)	Total time (s)
80 × 20 × 12	115200	22113	20	0.20	3.00	0.140	3.72
80 × 20 × 12	115200	22113	50	0.50	3.00	0.135	7.68
80 × 20 × 12	115200	22113	100	1.00	3.00	0.135	14.42
80 × 20 × 12	115200	22113	300	3.00	3.00	0.137	42.13

mesh. Since the DOF count remains fixed, the average time per step stays nearly constant, while the total runtime increases with the number of time steps. We also observed that for mesh discretizations over 400 × 100 × 60, with more than 2470561 DOFs the peak memory usage surges to 30-GB and simulations slow down substantially.

5.1 | Suitability for Data-Driven Applications

The time-stepping loop produces a sequence of nodal temperature fields, stored in `u_sol`, over the duration of the simulation. Each entry `u_sol[i]` contains the discrete temperature solution at time t_i. Together with the mesh, time grid, and process parameters,

these snapshots can serve as a useful basis for several data-driven studies:

- **Reduced-order modeling**: The snapshot data may be used to construct reduced bases, for example through proper orthogonal decomposition (POD), in order to investigate whether the thermal field admits a low-dimensional representation for the parameter ranges of interest.
- **Surrogate model training**: By varying process parameters such as laser power Q_p, scan speed v_s, and beam radius r, the framework can generate parametric datasets for training surrogate models, including neural-network and Gaussian-process-based regressors.
- **Physics-informed learning**: The governing equation Equation (1) and the boundary conditions may be incorporated as residual penalties or constraints in physics-informed learning frameworks trained on simulation data.
- **Toward digital twins**: With additional model reduction, calibration, and timing studies, the framework could support future work on online monitoring or control-oriented thermal models.

Because the implementation is Python-based and the solution data can be exported as NumPy arrays, it can be transferred with little effort to common machine-learning workflows, including those built on PyTorch or JAX.

5.2 | Limitations and Assumptions

A few simplifications have been made in the present formulation that should be acknowledged:

- **Constant material properties**: Both k and C_p are assumed independent of temperature. In reality, for metals near their melting point, these properties vary significantly and can exhibit discontinuous changes at phase transformation temperatures. Incorporating temperature-dependent properties would require modifying the governing equation to $\nabla \cdot (k(T)\nabla T)$, introducing additional nonlinearity into the bulk equation.
- **No phase change**: The present model does not account for latent heat associated with melting and solidification. In laser-based additive manufacturing, the melt pool temperature can exceed the solidus temperature, and the associated latent heat release/absorption has a strong effect on the thermal history. The enthalpy method or the apparent heat capacity method can be used to incorporate phase change within the present FEM framework.
- **Gaussian surface flux only**: The Goldak double-ellipsoid volumetric heat source model [11], which distributes heat within the material rather than on the surface, is more accurate for deep-penetration welds and high-power laser processes. The surface Gaussian model adopted here is appropriate for cases where the penetration depth is small relative to the element size.
- **No fluid dynamics in melt pool**: The simulation does not account for convective heat transfer within any molten material (Marangoni flow), which can significantly redistribute heat and alter the melt pool geometry in practice.

Nonetheless, this 99-line code serves as a concise and efficient implementation of the benchmark problem, featuring key components such as finite element assembly, nonlinear heat loss, Newton iteration, and AMG-preconditioned PCG solver. Advanced features like adaptive time stepping, adaptive mesh refinement, and broader solver interfaces may be added later as part of a more complete software framework.

6 | Conclusions

This work introduced a transparent, self-contained Python framework for simulating three-dimensional transient heat conduction with a moving Gaussian heat source. The exposition bridged the continuous mathematical formulation and its discrete computational implementation, covering the strong and weak forms, the θ-method for time discretization, and the nonlinear residual and consistent Jacobian calculation. The resulting approach is physically rigorous, mathematically transparent, and practical for both teaching and research purposes.

The principal contributions of this work are summarized as follows:

1. **Compact FEM implementation.** We developed a full three-dimensional, transient heat conduction solver—including nonlinear convective and radiative boundaries as well as a moving Gaussian laser source—in approximately 100 lines of Python utilizing `scikit-fem`. The `@LinearForm` and `@BilinearForm` decorators enable an explicit connection between the weak form and the corresponding code.
2. **Comprehensive mathematical derivation.** The article systematically derived the governing equations, Galerkin weak form, generalized θ-method, and Newton–Raphson linearization using a consistent Jacobian. As such, the framework serves as a valuable pedagogical reference for nonlinear finite element methods.
3. **Efficient nonlinear solution strategy.** The combination of Newton iteration, AMG-preconditioned PCG, and warm-starting from the previous time step yields a robust and efficient solver. In practice, convergence is typically achieved in 2–3 Newton iterations per time step, with highly efficient linear solves.
4. **Utility for data generation.** The framework is well-suited for generating high-fidelity three-dimensional temperature fields for use in reduced-order modeling, surrogate modeling, and physics-informed machine learning. Its modular design also facilitates systematic parametric studies.
5. **Reproducibility and accessibility.** Relying solely on freely available Python libraries and requiring no proprietary software, the framework delivers reproducible results ideal for teaching and open science. The code can be readily executed in Google Colab after installing the required packages (practically only two, as `numpy`/`scipy` come pre-installed).

Several promising avenues remain for future development, including temperature-dependent material properties, latent heat effects and phase change, alternative volumetric source models (e.g., Goldak's double ellipsoid), thermo-mechanical

coupling, adaptive mesh refinement, higher-order time discretization, and integration with machine learning frameworks for differentiable simulation and parameter optimization.

The complete source code appears in the Appendix as supplementary material. It is intended to support students and researchers in building thermal simulation tools and exploring connections between finite element methods and data-driven modeling in manufacturing contexts.

Acknowledgements

The author acknowledges Prof. Tuhin Mukherjee in the Department of Mechanical Engineering of Iowa State University for his inputs and the Texas A&M Institute of Data Science, where part of this study was performed.

Conflicts of Interest

The author declares no conflicts of interest.

Data Availability Statement

The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.

References

1. T. DebRoy, H. L. Wei, J. S. Zuback, et al., "Additive Manufacturing of Metallic Components - Process, Structure and Properties," *Progress in Materials Science* 92 (2018): 112–224.

2. L. E. Lindgren, "Numerical Modelling of Welding," *Computer Methods in Applied Mechanics and Engineering* 195, no. 48–49 (2006): 6710–6736.

3. N. E. Hodge, R. M. Ferencz, and J. M. Solberg, "Implementation of a Thermomechanical Model for the Simulation of Selective Laser Melting," *Computational Mechanics* 54 (2014): 33–51.

4. Z. Luo and Y. Zhao, "A Survey of Finite Element Analysis of Temperature and Thermal Stress Fields in Powder Bed Fusion Additive Manufacturing," *Additive Manufacturing* 21 (2018): 318–332.

5. O. C. Zienkiewicz, R. L. Taylor, and J. Z. Zhu, The Finite Element Method: Its Basis and Fundamentals, 7th ed. (Butterworth-Heinemann, 2013).

6. T. J. R. Hughes, *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis* (Dover Publications, 2012).

7. A. Logg, K. A. Mardal, and G. Wells, *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book* (Springer, 2012).

8. F. Rathgeber, D. A. Ham, L. Mitchell, et al., "Firedrake: Automating the Finite Element Method by Composing Abstractions," *ACM Transactions on Mathematical Software* 43, no. 3 (2016): 1–27.

9. R. Cimrman, V. Lukeš, and E. Rohan, "Multiscale Finite Element Calculations in Python Using Sfepy," *Advances in Computational Mathematics* 45 (2019): 1897–1921.

10. T. Gustafsson and G. D. McBain, "Scikit-Fem: A Python Package for Finite Element Assembly," *Journal of Open Source Software* 5, no. 52 (2020): 2369.

11. J. Goldak, A. Chakravarti, and M. Bibby, "A New Finite Element Model for Welding Heat Sources," *Metallurgical Transactions B* 15, no. 2 (1984): 299–305.

12. J. Irwin and P. Michaleris, "A Line Heat Input Model for Additive Manufacturing," *Journal of Manufacturing Science and Engineering* 138, no. 11 (2016): 111004.

13. N. Bell, L. N. Olson, J. Schroder, and B. Southworth, "Pyamg: Algebraic Multigrid Solvers in Python," *Journal of Open Source Software* 8, no. 87 (2023): 5495.

14. A. Fic, R. A. Bialecki, and A. J. Kassab, "Solving Transient Nonlinear Heat Conduction Problems by Proper Orthogonal Decomposition and the Finite-Element Method," *Numerical Heat Transfer, Part B: Fundamentals* 48, no. 2 (2005): 103–124.

15. P. Benner, S. Gugercin, and K. Willcox, "A Survey of Projection-Based Model Reduction Methods for Parametric Dynamical Systems," *SIAM Review* 57, no. 4 (2015): 483–531.

16. C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning* (Cambridge, MA: MIT Press, 2006).

17. G. Tapia, S. A. Khairallah, M. J. Matthews, W. E. King, and A. Elwany, "Gaussian Process-Based Surrogate Modeling Framework for Process Planning in Laser Powder-Bed Fusion Additive Manufacturing of 316L Stainless Steel," *International Journal of Advanced Manufacturing Technology* 94, no. 9–12 (2018): 3591–3603.

18. M. Raissi, P. Perdikaris, and G. E. Karniadakis, "Physics-Informed Neural Networks: A Deep Learning Framework for Solving Forward and Inverse Problems Involving Nonlinear Partial Differential Equations," *Journal of Computational Physics* 378 (2019): 686–707.

19. S. Liao, T. Xue, J. Jeong, S. Webster, K. Ehmann, and J. Cao, "Hybrid Thermal Modeling of Additive Manufacturing Processes Using Physics-Informed Neural Networks for Temperature Prediction and Parameter Identification," *Computational Mechanics* 72, no. 3 (2023): 499–512.

20. A. Paszke, S. Gross, F. Massa, et al., "Pytorch: An Imperative Style, High-Performance Deep Learning Library," *In Advances in 794 Neural Information Processing Systems* 32 (2019.

21. J. Bradbury, R. Frostig, P. Hawkins, et al. JAX: Composable Transformations of Python+NumPy Programs, 2018, http://github.com/jax-ml/jax.

22. F. P. Incropera, D. P. DeWitt, T. L. Bergman, and A. S. Lavine, Fundamentals of Heat and Mass Transfer, 7th ed. (John Wiley & Sons, 2011).

23. R. J. Guyan, "Reduction of Stiffness and Mass Matrices," *AIAA Journal* 3, no. 2 (1965): 380.

24. A. Ern and J. L. Guermond, *Theory and Practice of Finite Elements* (Springer, 2004).

25. G. Strang and G. Fix, *An Analysis of the Finite Element Method* (Prentice-Hall, 1973).

26. M. Javani, Y. Kiani, and M. R. Eslami, "Geometrically Nonlinear Rapid Surface Heating of Temperature-Dependent Fgm Arches," *Aerospace Science and Technology* 90, no. 1 (2019): 264–274.

27. M. Javani, Y. Kiani, and M. R. Eslami, "Rapid Heating Vibrations of Fgm Annular Sector Plates," *Engineering with Computers* 37, no. 1 (2021): 305–322.

28. A. Keibolahi, Y. Kiani, and M. R. Eslami, "Dynamic Snap-Through of Shallow Arches under Thermal Shock," *Aerospace Science and Technology* 77, no. 1 (2018): 545–554.

29. M. Javani, Y. Kiani, and M. R. Eslami, "Nonlinear Axisymmetric Response of Temperature-Dependent Fgm Conical Shells under Rapid Heating," *Acta Mechanica* 230, no. 9 (2019): 3019–3039.

30. H. R. Esmaeili, H. Arvin, and Y. Kiani, "Axisymmetric Nonlinear Rapid Heating of Fgm Cylindrical Shells," *Journal of Thermal Stresses* 42, no. 4 (2019): 490–505.

31. C. R. Harris, K. J. Millman, S. J. van der Walt, et al., "Array Programming With Numpy," *Nature* 585, no. 7825 (2020): 357–362.

32. P. Virtanen, R. Gommers, T. E. Oliphant, et al., "Scipy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods* 17 (2020): 261–272.

33. S. Liao, T. Xue, J. Jeong, S. Webster, K. Ehmann, and J. Cao, "Hybrid Thermal Modeling of Additive Manufacturing Processes Using Physics-Informed Neural Networks for Temperature Prediction and Parameter Identification," *Computational Mechanics* 72, no. 3 (2023): 499–512.

Appendix A

Complete scikit-fem Implementation

For completeness, the full scikit-fem implementation used in this work is provided below as a single uninterrupted listing. The code is written in exactly 99 lines.

```
import numpy as np
from pyamg import smoothed_aggregation_solver
from skfem import *
from skfem.helpers import grad, dot

lx, ly, lz = 40.0, 10.0, 6.0
nx, ny, nz = 2 * int(lx), 2 * int(ly), 2 * int(lz)
rho, cp, k = 8e-3, 0.5, 0.01
T_infty, h = 298.0, 2e-5
Rboltz, emiss = 5.6704e-14, 0.3
scan_speed, r, Qp, eta = 10.0, 1.5, 500.0, 0.4
dt, theta, t_end = 0.01, 0.5, 3.0
ts = np.arange(0.0, t_end + dt, dt)
save_every = 1

mesh = MeshTet.init_tensor(
    np.linspace(0, lx, nx + 1),
    np.linspace(0, ly, ny + 1),
    np.linspace(0, lz, nz + 1),
).with_boundaries({
    "left": lambda x: np.isclose(x[0], 0.0),
    "right": lambda x: np.isclose(x[0], lx),
    "front": lambda x: np.isclose(x[1], 0.0),
    "back": lambda x: np.isclose(x[1], ly),
    "bottom": lambda x: np.isclose(x[2], 0.0),
    "top": lambda x: np.isclose(x[2], lz),
})

element = ElementTetP1() #For higher accuracy convert to ELementP2()
basis = Basis(mesh, element)
bottom_dofs = basis.get_dofs("bottom").all()
facets = np.unique(np.concatenate([mesh.boundaries[n] for n in ("left",
     "right", "front", "back", "top")]))
fb = FacetBasis(mesh, element, facets=facets)
ft = FacetBasis(mesh, element, facets=mesh.boundaries["top"])
laser_t = {"time": 0.0}

@BilinearForm
def transient_term(u, v, w): return u * v

@BilinearForm
def diffusion_term(u, v, w): return dot(grad(u), grad(v))

@LinearForm
def qloss(v, p):
    T = (1.0 -theta) * p["prev"] + theta * p["trial"]
    return (h * (T -T_infty) + Rboltz * emiss * (T**4 -T_infty**4)) * v

@BilinearForm
def jloss(u, v, p):
    T = (1.0 -theta) * p["prev"] + theta * p["trial"]
    return theta * (h + 4.0 * Rboltz * emiss * T**3) * u * v

```

```
        @LinearForm
        def laser(v, w):
            xc, yc = 0.1 * lx + scan_speed * laser_t["time"], ly / 2.0
            rsq = (w.x[0] -xc) ** 2 + (w.x[1] -yc) ** 2
            return (2.0 * Qp * eta / (np.pi * r**2)) * np.exp(-2.0 * rsq / r
                **2) * v

        M = asm(transient_term, basis).tocsr()
        K = asm(diffusion_term, basis).tocsr()
        mfac = rho * cp / dt
        A0 = mfac * M + theta * k * K
        B0 = ((1.0 -theta) * k * K -mfac * M).tocsr()

        def assemble(u, u_old, t_theta):
            laser_t["time"] = t_theta
            ub, uob = fb.interpolate(u), fb.interpolate(u_old)
            r = A0 @ u + B0 @ u_old
            r += asm(qloss, fb, trial=ub, prev=uob)
            r -= asm(laser, ft)
            J = A0 + asm(jloss, fb, trial=ub, prev=uob)
            return J, r
        # Newton solver
        def newton(u, u_old, t_theta, ml=None, tol=1e-8, maxit=20):
            u = u.copy()
            u[bottom_dofs] = T_infty
            for _ in range(maxit):
                J, r = assemble(u, u_old, t_theta)
                A, b, x, I = condense(J, -r, D=bottom_dofs)
                A = A.tocsr()
                ml = smoothed_aggregation_solver(A) if ml is None else ml
                if hasattr(ml, "change_solve_matrix"): ml.change_solve_matrix(A)
                du_red = solve(A, b, solver=solver_iter_pcg(
                    M=ml.aspreconditioner(cycle="V"), rtol=1e-6, atol=1e-8,
                        verbose=False))
                du = x.copy()
                du[I] = du_red
                u_new = u + du
                u_new[bottom_dofs] = T_infty
                if np.linalg.norm(u_new -u) < tol: return u_new, ml
                u = u_new
            raise RuntimeError("Newton solver did not converge.")
        # Time stepping
        u_old = np.full(basis.N, T_infty)
        u_sol, ml = [u_old.copy()], None
        for n in range(len(ts) -1):
            t_theta = (1.0 -theta) * ts[n] + theta * ts[n + 1]
            u_old, ml = newton(u_old, u_old, t_theta, ml=ml)
            if (n + 1)%
            print(f"step={n+1:03d}, time={ts[n+1]:.4f}, Tmax={u_old.max():.6f}"
                )
```

Listing 7: Complete 99-line scikit-fem implementation for the moving heat-source problem.

Appendix B

Second-Order Accuracy of the Crank-Nicolson Method

We prove that the Crank-Nicolson (CN) scheme applied to the 1D heat equation is second-order accurate in both time and space, i.e., the local truncation error (LTE) is $\mathcal{O}(\Delta t^2) + \mathcal{O}(\Delta x^2)$.

Model Problem

Consider the 1D heat equation

$$u_t = \alpha u_{xx}, x \in [0, L], t > 0. \tag{B1}$$

Let $u_j^n \approx u(x_j, t_n)$ with uniform grid spacing Δx and time step Δt. The general θ-scheme reads

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = \alpha\left[\theta\delta_{xx}u_j^{n+1} + (1-\theta)\delta_{xx}u_j^n\right], \tag{B2}$$

where $\delta_{xx}u_j^n = \left(u_{j+1}^n - 2u_j^n + u_{j-1}^n\right)/\Delta x^2$. The Crank-Nicolson method corresponds to $\theta = \frac{1}{2}$.

Local Truncation Error Via Taylor Expansion

Substituting the exact solution u into (B2), the LTE is

$$\tau_j^n = \frac{u(x_j, t_{n+1}) - u(x_j, t_n)}{\Delta t} - \alpha[\theta\delta_{xx}u(x_j, t_{n+1}) + (1-\theta)\delta_{xx}u(x_j, t_n)]. \tag{B3}$$

B.2.1 | *Time Derivative*

Taylor expansion of $u(x_j, t_{n+1})$ about t_n gives

$$\frac{u(x_j, t_{n+1}) - u(x_j, t_n)}{\Delta t} = u_t + \frac{\Delta t}{2}u_{tt} + \frac{\Delta t^2}{6}u_{ttt} + \mathcal{O}(\Delta t^3). \tag{B4}$$

B.2.2 | *Spatial Terms*

The central-difference operator satisfies

$$\delta_{xx}u_j^n = u_{xx}|_{t_n} + \frac{\Delta x^2}{12}u_{xxxx}|_{t_n} + \mathcal{O}(\Delta x^4). \tag{B5}$$

Expanding the level-$(n+1)$ spatial term about t_n:

$$\delta_{xx}u_j^{n+1} = u_{xx}|_{t_n} + \Delta t u_{xxt}|_{t_n} + \frac{\Delta t^2}{2}u_{xxtt}|_{t_n} + \frac{\Delta x^2}{12}u_{xxxx}|_{t_n} + \mathcal{O}(\Delta x^4, \Delta t^3). \tag{B6}$$

The θ-weighted average of the two spatial terms becomes

$$\theta\delta_{xx}u_j^{n+1} + (1-\theta)\delta_{xx}u_j^n = u_{xx}|_{t_n} + \theta\Delta t u_{xxt}|_{t_n} + \frac{\theta\Delta t^2}{2}u_{xxtt}|_{t_n} + \frac{\Delta x^2}{12}u_{xxxx}|_{t_n} + \mathcal{O}(\Delta x^4, \Delta t^3). \tag{B7}$$

Assembling the LTE

Substituting (B4) and (B7) into τ_j^n, and using the PDE (B1) (so that $u_t - \alpha u_{xx} = 0$) and the consistency relations $u_{tt} = \alpha u_{xxt}$, $u_{ttt} = \alpha^2 u_{xxxx}$, the LTE simplifies to

$$\tau_j^n = \underbrace{\left(\frac{1}{2} - \theta\right)\Delta t\alpha u_{xxt}}_{\text{first-order term}} - \frac{\Delta x^2}{12}\alpha u_{xxxx} + \mathcal{O}(\Delta t^2, \Delta x^4). \tag{B8}$$

Cancellation at $\theta = \frac{1}{2}$

Setting $\theta = \frac{1}{2}$ (Crank-Nicolson), the coefficient of the $\mathcal{O}(\Delta t)$ term in (B8) vanishes identically, leaving

$$\boxed{\tau_j^n = -\frac{\alpha}{12}\Delta x^2 u_{xxxx} - \frac{\alpha^2}{12}\Delta t^2 u_{xxxx} + \mathcal{O}(\Delta t^4, \Delta x^4) = \mathcal{O}(\Delta t^2) + \mathcal{O}(\Delta x^2).} \tag{37}$$

Hence the Crank-Nicolson scheme is *second-order accurate* in both time and space.

Comparison with Other θ-Schemes

Table B1

TABLE B1 | Temporal and spatial accuracy of the θ-scheme for the heat equation.

θ	Scheme	Time accuracy	Space accuracy
0	Forward Euler	$\mathcal{O}(\Delta t)$	$\mathcal{O}(\Delta x^2)$
1	Backward Euler	$\mathcal{O}(\Delta t)$	$\mathcal{O}(\Delta x^2)$
$\frac{1}{2}$	**Crank-Nicolson**	$\mathcal{O}(\Delta t^2)$	$\mathcal{O}(\Delta x^2)$

Only at $\theta = \frac{1}{2}$ does the leading $\mathcal{O}(\Delta t)$ coefficient in (B8) vanish, which confirms that Crank-Nicolson is the unique second-order member of the θ-family.

Appendix C

Stability of the Crank-Nicolson Method

The stability of the Crank-Nicolson (CN) scheme is analyzed using von Neumann (Fourier) analysis. We show that the CN method is *unconditionally stable* for the 1D heat equation, meaning the numerical solution does not grow without bound for any choice of Δt and Δx.

C.1 | *Von Neumann Stability Analysis*

Consider the θ-scheme applied to the heat equation:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = \alpha\left[\theta\delta_{xx}u_j^{n+1} + (1-\theta)\delta_{xx}u_j^n\right]. \tag{C1}$$

Assume a Fourier mode $u_j^n = G^n e^{ikx_j}$, where G is the amplification factor and k is the wavenumber. Substituting and using

$$\delta_{xx}(e^{ikx_j}) = \frac{e^{ik\Delta x} - 2 + e^{-ik\Delta x}}{\Delta x^2}e^{ikx_j} = \lambda e^{ikx_j}, \lambda = \frac{2(\cos(k\Delta x) - 1)}{\Delta x^2} \le 0, \tag{C2}$$

and dividing through by $G^n e^{ikx_j}$, the scheme reduces to

$$\frac{G-1}{\Delta t} = \alpha[\theta\lambda G + (1-\theta)\lambda]. \tag{C3}$$

Solving for G with $\mu := \alpha\Delta t\lambda \le 0$:

$$G = \frac{1 + (1-\theta)\mu}{1 - \theta\mu}. \tag{C4}$$

For the Crank-Nicolson method ($\theta = \frac{1}{2}$), (C4) becomes

$$G_{\text{CN}} = \frac{1 + \frac{\mu}{2}}{1 - \frac{\mu}{2}}. \tag{C5}$$

C.2 | *Proof of Unconditional Stability*

Since $\lambda \le 0$ and $\alpha, \Delta t > 0$, we have $\mu \le 0$. Write $\mu = -\rho$ with $\rho \ge 0$. Then (C5) gives

$$G_{\mathrm{CN}} = \frac{1 - \frac{\rho}{2}}{1 + \frac{\rho}{2}}, \tag{C6}$$

so

$$|G_{\mathrm{CN}}|^2 = \frac{\left(1 - \frac{\rho}{2}\right)^2}{(1 + \frac{\rho}{2})^2} \le 1 \;\Leftrightarrow\; \left(1 - \frac{\rho}{2}\right)^2 \le \left(1 + \frac{\rho}{2}\right)^2 \;\Leftrightarrow\; 0 \le 2\rho, \tag{C7}$$

which holds for all $\rho \ge 0$. Therefore $|G_{\mathrm{CN}}| \le 1$ unconditionally, and the scheme is stable for *any* Δt and Δx. □

C.3 | *Comparison with Other θ-Schemes*

- **Forward Euler** ($\theta = 0$): $G = 1 + \mu$. Requires $\mu \ge -2$, i.e., $\Delta t \le \frac{\Delta x^2}{2\alpha}$ (conditionally stable).
- **Backward Euler** ($\theta = 1$): $G = \frac{1}{1-\mu} = \frac{1}{1+\rho} \in (0, 1]$. Unconditionally stable; all modes are monotonically damped.
- **Crank-Nicolson** ($\theta = \frac{1}{2}$): $|G| \le 1$ unconditionally. Modes are damped but, for large ρ, $G \to -1$, meaning high-frequency modes can oscillate in sign between time steps (see Remarks below).

C.4 | *Remarks*

Although Crank-Nicolson is unconditionally stable ($|G| \le 1$), it is only *weakly dissipative*: for large time steps or high-frequency modes ($\rho \gg 1$), the amplification factor approaches $G \approx -1$. This causes high-frequency Fourier modes to flip sign at every time step, producing spurious oscillations in the numerical solution near sharp gradients or discontinuities. Backward Euler ($\theta = 1$) strongly damps these modes ($G \to 0$ as $\rho \to \infty$), so it suppresses such oscillations at the cost of increased numerical dissipation. In practice, moderate time steps are recommended for Crank-Nicolson, or a small amount of upward θ-blending (e.g., $\theta = 0.55$–0.6) is used to damp spurious oscillations while retaining near-second-order accuracy.