FastFEM v0.0.1

A Python package for solving PDEs with the finite element method

fastfem.com

Sina Atalay[1*], Sacha Escudier[1*], Kentaro Hanson[1*]

[1]Princeton University, Princeton, NJ, USA

[*]All authors contributed equally

December 2024

Abstract

The finite element method is one of the most widely used techniques for solving partial differential equations in various industries. There are numerous open-source software packages available that implement the finite element method. However, advancements continue to be made, particularly with the emergence of new array computing libraries such as JAX and PyTorch, which feature automatic differentiation capabilities. Additionally, there is still significant potential for innovation in the user and application interfaces of finite element software to make them more accessible for researchers. `FastFEM` is an open-source finite element software designed to provide a clean and easy-to-use Python interface while opening the possibility to leverage JAX (in a future version) for computations with its automatic differentiation features. This report presents the capabilities of `FastFEM v0.0.1`, including its mesher, solver, and plotter, along with example code snippets. The underlying theory of the finite element method is summarized, and the implementation within `FastFEM` is briefly discussed. Furthermore, the strategies employed and tools utilized to maintain high software quality are explained. Finally, conclusions are drawn, and an outlook for the future of `FastFEM` is provided.

1 Introduction

Partial differential equations (PDEs) are the fundamental tools for mathematically modeling natural phenomena. Many fundamental phenomena observed in nature, such as general relativity[1], quantum mechanics[2], heat diffusion[3], fluid mechanics[4], pricing of financial derivative contracts[5], structural analysis[6], and electromagnetism[7], are described by PDEs. However, most of these PDEs do not have closed-form solutions, especially in complex geometries. Therefore, engineers have developed many numerical methods for solving PDEs. One of the most popular numerical methods among them is the finite-element method (FEM), which originated in the early 1940s[8]. FEM is capable of solving non-linear PDEs in highly complex geometries. Since the 1940s, FEM has undergone significant advancements and has revolutionized the way scientific modeling and engineering design. Today, it is widely used in many industrial applications.

There are many open-source FEM software options available today[9–16]. However, we think there is significant potential for innovation, particularly in user and application interfaces. `FastFEM`[17] is developed with the goal of

- Creating an easy-to-use and clean Python interface to make FEM more accessible to researchers
- Using modern tools like JAX[18] for advanced array computing with automatic differentiation capabilities

One of the motivations for creating a Python interface was the capability of the language to create very intuitive-to-use interfaces. A modern Python interface can offer users a great way of describing FEM problems. The other motivation was leveraging the existing scientific Python environment. Python's popularity in the scientific world is still increasing, and modern libraries with state-of-the-art technologies like JAX, PyVista[19], etc., are being developed.

FastFEM is planned to be a big project, but as the goal of `v0.0.1`, we decided to focus on 2D parabolic PDEs. Parabolic PDEs, such as the heat diffusion equation, Poisson's equation, and the Black-Scholes equation, are very common in various applications. A 2-dimensional parabolic PDE can be expressed as

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

where $f(x,y,t)$, $h(f)$, and $g(x,y)$ are scalar functions, x and y are spatial coordinates, and t is time.

Currently, `FastFEM v0.0.1` can

1. create a 2D mesh,
2. take initial conditions, $h(f)$, and $g(x,y)$ as inputs,
3. solve Equation 1 with FEM, and
4. plot the solution.

In this report, the theory behind FEM is summarized. The features and capabilities of `FastFEM v0.0.1` are presented. Some example results are shown. Finally, the conclusion and outlook are discussed.

2 Theory

We hereby present an overview of the mathematical formalism behind the finite element method. Let there be a parabolic partial differential equation of the form:

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = k(f)\frac{\partial f}{\partial t} + g(x,y) \tag{2}$$

Where f represents the function of interest (which may be interpreted as temperature in a heat conduction problem, for example), $k(f)$ is some function of f and $g(x,y)$ is some spatial source/sink term. In a more compact form, this may be written as:

$$\nabla^2 f = k\frac{\partial f}{\partial t} + g(x,y) \tag{3}$$

And may further be formulated in its weak form, where equation 3 is multiplied by some test function v in a Sobolev space H_0^1, i.e. smoothly differential once.

$$\int_\Omega \nabla^2 f v dV = \int_\Omega k \frac{\partial f}{\partial t} v dV + \int_\Omega g v dV \tag{4}$$

For some domain of interest Ω, effectively determined by the initial and boundary conditions of the problem. Integrating equation 4 by parts and applying Dirichlet BC's for the test function v, we arrive at the following form.

$$-\int_\Omega \nabla \mathbf{f} \cdot \nabla \mathbf{v} dV = \int_\Omega k \frac{df}{dt} v + \int_\Omega g v dV \tag{5}$$

where each quantity may be represented as a sum:

$$f = \sum_i^n f_i \phi_i(x, y)$$
$$\dot{f} = \sum_i^n f_i' \phi_i(x, y)$$
$$g = \sum_i^n g_i \phi_i(x, y)$$
$$v = \sum_i^n v_i \phi_i(x, y)$$

For some shape functions $\phi_i(x, y)$, which constitute an appropriate choice of basis for the original PDE.

Then, the weak formulation is further simplified:

$$-\sum_i \sum_j \int_\Omega \nabla(f_i \phi_i) \cdot \nabla(v_j \phi_j) dV = \sum_i \sum_j \int_\Omega k(f_i' \phi_i)(v_j \phi_j) + \sum_i \sum_j \int_\Omega (g_i \phi_i)(v_j \phi_j) dV \tag{6}$$

$$-\sum_i \sum_j f_i v_j \int_\Omega (\nabla \phi_i) \cdot (\nabla \phi_j) dV = \sum_i \sum_j k f_i' v_j \int_\Omega \phi_i \phi_j + \sum_i \sum_j g_i v_j \int_\Omega \phi_i \phi_j dV \tag{7}$$

Where equation 7 may be expressed in matrix form as:

$$K\mathbf{f} = M(\mathbf{g} + \dot{f}) \tag{8}$$

Where K is the stiffness matrix, M is the mass matrix, and are expressed as:

$$M(\phi_i, \phi_j) = \int_\Omega \phi_i \phi_j \, dV \tag{9}$$

$$K(\phi_i, \phi_j) = \int_\Omega (\nabla \phi_i) \cdot (\nabla \phi_j) \, dV \tag{10}$$

Then, equation 8 is a linear system of equations, and may be solved using any scheme appropriate scheme.

```python
import fastfem.mesh as m

square_mesh = (
    m.create_a_square_mesh(
        side_length=1.0,
        nodes_in_horizontal_direction=10,
        nodes_in_vertical_direction=10,
        element_type="quadrangle",
    ),
)

create_a_rectangle_mesh = (
    m.create_a_rectangle_mesh(
        horizontal_length=1.0,
        vertical_length=0.5,
        nodes_in_horizontal_direction=10,
        nodes_in_vertical_direction=10,
        element_type="triangle",
    ),
)
```

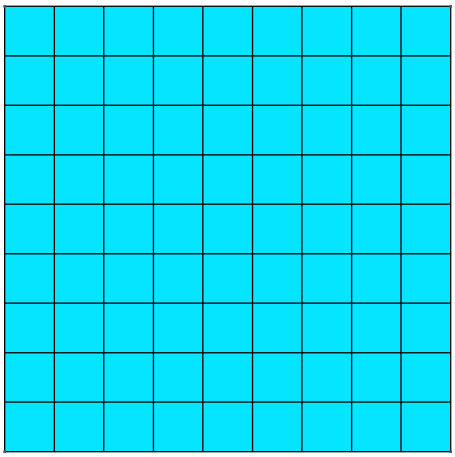
Figure 1: Two available functions of `FastFEM v0.0.1` for generating 2D square and rectangle meshes that are shown in Figure 2.

3 Features and Capabilities

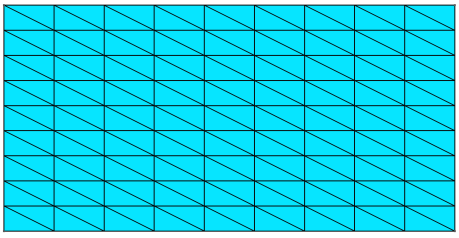
This section summarizes `FastFEM v0.0.1`'s features and capabilities: mesh generation, solving parabolic PDEs with FEM using elements and fields, and plotting.

3.1 Mesher

`FastFEM` uses Gmsh[20] for mesh generation. Currently, `v0.0.1` offers two fundamental 2D mesh generation functions, as shown in Figure 1. The results of the functions are shown in Figure 2. `FastFEM` is fully statically typed so that users can get autocomplete suggestions when writing the function's arguments and using the mesh object it returns.



(a) Square mesh.



(b) Rectangle mesh.

Figure 2: Two examples of mesh generated with the functions in Figure 1.

```python
import fastfem.mesh as m

points = [
    m.Point(0, 0),
    m.Point(1, 0),
    m.Point(1, 1),
    m.Point(0.5, 1.5),
    m.Point(0, 1),
    m.Point(-0.5, 0.5),
    m.Point(0, 0),
]
lines = [m.Line(points[i], points[i + 1], number_of_nodes=10) for i in range(6)]
surface = m.Surface(lines, element_type="quadrangle")
mesh = m.mesh()
```

Figure 3: The code that generates the mesh of an arbitrary geometry shown in Figure 4.

`FastFEM v0.0.1` also provides a general-purpose, object-oriented interface for 2D mesh generation by abstracting Gmsh's functional approach. For example, a mesh of an arbitrary 2D geometry, as shown in Figure 4, can be created using the code presented in Figure 3.

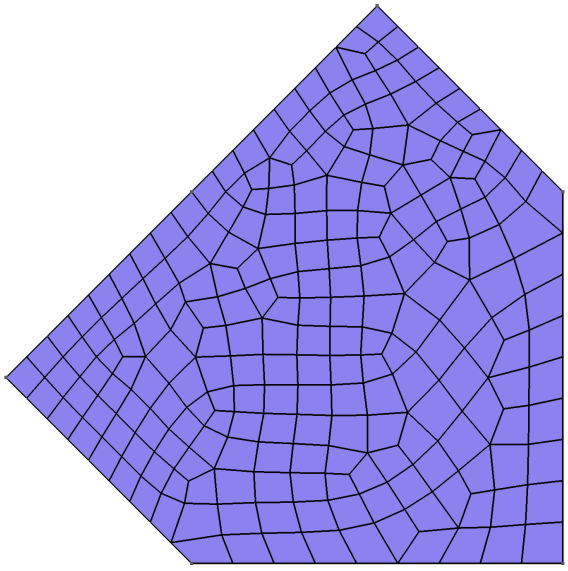


Figure 4: An arbitrary 2D mesh generated using the code in Figure 3.

3.2 Elements and Fields

In order to use the weak formulation, `FastFEM` needs to know how to integrate functions on a given mesh. Each element type specifies both the shape functions and the algorithm used to integrate a function on that element. For example, the mass matrix entries (9) on two different element types are:

- For linear triangular elements with shape functions $\{\phi_1, \phi_2, \phi_3\}$ having the Kronecker delta property for each vertex ($\phi_i(v_j) = \delta_{ij}$ for vertices v_1, v_2, v_3), the analytical solution to the integral is known, since the shape functions are linear.
- Spectral elements, a technique used in fluid dynamics (Patera 1984)[21] and other

continuum mechanical fields, have shape functions $\phi_{i_1,i_2}(F(x,y)) = L_{i_1}(x)L_{i_2}(y)$ for Lagrange interpolation polynomials L_i based on GLL (Gauss-Lobatto-Legendre) quadrature[22], where F denotes the coordinate transform from a reference space $[-1,1]^2$ to the element in the mesh's space. Integration is done using GLL quadrature in order to produce a diagonal mass matrix, so the integral is computed

$$\begin{aligned}\int_{\Omega_{SE}} \phi_{i_1,i_2}\phi_{j_1,j_2}\ dV &= \int_{\Omega_{SE}} L_{i_1}(x)L_{i_2}(y)L_{j_1}(x)L_{j_2}(y)\ dV \\ &\approx \sum_{k_1,k_2=0}^{N} \alpha_{k_1}\alpha_{k_2} J(F(t_{k_1},t_{k_2}))L_{i_1}(t_{k_1})L_{i_2}(t_{k_2})L_{j_1}(t_{k_1})L_{j_2}(t_{k_2}), \\ &= \alpha_{i_1}\alpha_{i_2} J(F(t_{k_1},t_{k_2}))\delta_{i_1,j_1}\delta_{i_2,j_2}\end{aligned} \tag{11}$$

where $J = \rho|\det DF|$ is the Jacobian with some scaling parameter ρ, which may vary in space. This is relevant for PDEs with a varying mass term.

3.2.1 Elements

To employ the different behaviors, elements inherit from a superclass that with abstract methods for each integral. For H^1 elements in 2D, these are

```
element.integrate_field(position_field, field, jacobian_scale)
```

$$\int_{\Omega_E} f\ dV \tag{12}$$

```
element.integrate_basis_times_field(position_field, field, indices, jacobian_scale)
```

$$\int_{\Omega_E} \phi_i f\ dV \tag{13}$$

```
element.integrate_grad_basis_dot_field(position_field, field, indices, jacobian_scale)
```

$$\int_{\Omega_E} (\nabla\phi_i)\cdot\mathbf{f}\ dV \tag{14}$$

```
element.integrate_grad_basis_dot_grad_field(position_field, field, indices, jacobian_scale)
```

$$\int_{\Omega_E} (\nabla\phi_i)\cdot(\nabla f)\ dV \tag{15}$$

`indices` specify which basis functions should be computed. This is helpful for spectral elements, where only diagonal basis entries are needed. These functions implement integration against a field instead of two shape functions. This is because it may be computationally efficient to compute the contracted values $(I_{ij}u^j)_i$ instead of computing the matrix first, then contracting. For example, the stiffness matrix entries (10) are never used directly for the heat equation (1), but instead are used in contraction with the heat field f. This contraction is to produce the integral (15), which can just be called by itself. If uncontracted matrix entries are desired, one can easily integrate each basis function individually. As of writing boundary integrals have not yet been implemented, so only homogeneous boundary conditions can be solved for (Dirichlet or Neumann).

In addition, a `mass_matrix` method is implemented to provide a one-line code to obtain the mass matrix (9). These methods, along with the mesh element (3.2.3), provide the backbone to solving the weak form of various PDEs, like in Figure 5.

3.2.2 Fields

The fields passed into the integrate functions need to have an array structure, with indices for both the basis function index and tensor index. For example, the deformation gradient DF is represented in memory as coefficients $\left(\partial_k F^j\right)_i$, where

$$(DF)^j{}_k = \sum_i \left(\partial_k F^j\right)_i \phi_i, \quad F = F^j \mathbf{e}_j \tag{16}$$

when it is in the same space as the shape functions. Some elements may function better with multi-indexed shape functions (such as spectral elements), which further complicate the coefficient indices. Here, the field is represented with two sets of indices: the tensor indices and the shape indices. This combined with the *field stacking* feature we discuss below to vectorize function calls, introduces the need to keep track of which index corresponds to what.

While the size of the shape function multi-indices were fixed, the stack and tensor indices could vary and possibly collide. This became a problem when dealing with multiple field inputs, where indices might not line up due to implied broadcasting. Instead of forcing the user to make sure all of the indices line up, we decided that it was important to offload that burden, and let *numpy* broadcasting rules dictate the indices within each multi-index. However, having the user pass in index shapes or offsets as arguments bloated the function call and, while re-indexing was automatically managed, still had users keep track of index shapes.

To fix this, we utilize a `Field` wrapper class, which takes a *numpy* or *JAX* array with the corresponding tensor, shape function, and stack index shapes. This has the added benefit of allowing the delegation of operations to *JAX* or *numpy* depending on if the *JAX* feature set is necessary (say, for *autodiff*).

The Field Object Each field has three shapes corresponding to it:

- `point_shape` - The shape of the (pointwise) tensor the field represents.
- `basis_shape` - The shape of the basis indices used to reference the shape functions.
- `stack_shape` - The shape of the *field stack*, allowing for vectorized operations.

In addition, we store whether or not to use *JAX* functions. If `use_jax` is true for a field, then functions that delegate to `numpy` call `jax.numpy` instead. Additionally, any field resulting from an operation will inherit `use_jax` from any argument that has `use_jax == True`.

Field Stacking Due to the overhead of Python, vectorizing operations is important for performance. `FastFEM` does this by *field stacking*, so that a single operation can be performed on multiple fields at the same time. This is important for integrating on meshes (3.2.3), where multiple integration calls must be made on one type of element.

3.2.3 The Mesh Element

In addition to triangular and spectral elements, we have triangular mesh elements, which take a mesh from the mesher (3.1) and generate an element. The corresponding basis is

```
inv_mass_matrix = fnp.linalg.inv(
    fnp.moveaxis(
        mesh.mass_matrix(), (ShapeComponent.STACK, 0), (ShapeComponent.BASIS, 1)
    ),
    ShapeComponent.BASIS,
)


def solve_accel(wavefield):
    return -fnp.einsum(
        ShapeComponent.BASIS,
        "ij,j->i",
        inv_mass_matrix,
        mesh.integrate_grad_basis_dot_grad_field(wavefield, jacobian_scale=c2),
    )


Uddot = solve_accel(U)
while t < tmax:
    # use Newmark-beta
    U += dt * Udot + (0.5 * dt**2) * Uddot
    Udot += 0.5 * dt * Uddot
    Uddot = solve_accel(U)
    Udot += 0.5 * dt * Uddot
    t += dt
```

Figure 5: A snippet of the wave equation demo code, whose output can be found in Figure 6. The full code is found in `examples/demo_wave_equation.py`. The two integrals used here are the mass matrix terms and the stiffness matrix terms, which are computed using the corresponding functions, detailed in (3.2.1).

one dimensional, where each ϕ_i corresponds to a node (vertex) of the mesh. Each sub-element Ω_k corresponds to a polygon of the mesh. With vertices i_1, i_2, i_3, we relate the basis on the sub-element $\{\tilde{\phi}_1, \tilde{\phi}_2, \tilde{\phi}_3\}$ to the basis on the mesh element by:

$$\phi_{i_j}\big|_{\Omega_k} = \tilde{\phi}_j, \qquad j = 1, 2, 3 \tag{17}$$

Integrating a field on the mesh element is done as follows:

1. Create a new field with the sub-element's basis shape and an axis of size N prepended to the stack shape, where N is the number of sub-elements in the mesh. Values are populated according to the assembly rule (17).

2. Call the corresponding sub-element's integration function. Let I_k denote the result on sub-element k.

3. Perform an atomic addition. In the case of `element.integrate_field`, this is just an addition reduction. For basis integrals $I_k(\tilde{\phi}_j)$, the coefficient to ϕ_{i_j} is accumulated.

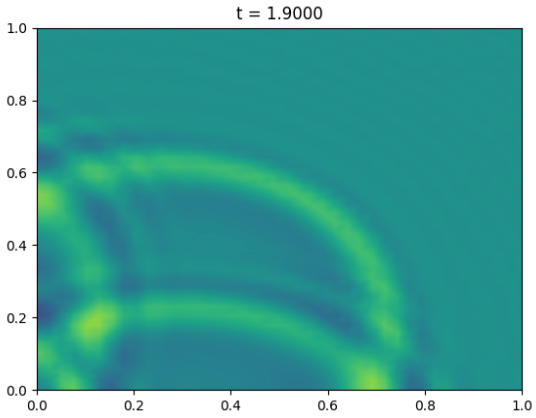


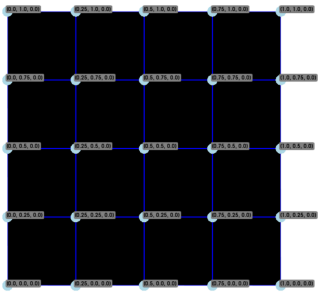
Figure 6: Output of (`examples/demo_wave_equation.py`). A single frame of the wave equation demo code. The relevant solving code is provided in Figure 5.

3.3 Plotter

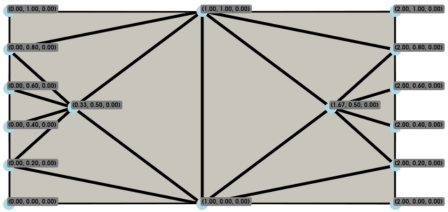
`FastFEM` uses `PyVista`'s plotting capabilities for visual representation. There are currently six methods available to the user:

- `define_plotter()`: This is a helper method, responsible for interfacing the mesh object as produced by the mesher to `PyVista`'s own requirements. Notably, it returns a PyVista grid object containing all relevant information on the nodes and connectivity of the mesh, and is subsequently used for plotter generation.
- `plot_mesh()`: This method is responsible for plotting the mesh object, and confirming `PyVista`'s redering is correct. In addition, it has aethetics related parameters that can be switched to the user's liking.
- `plot_data()`: This method will plot the data on a mesh, without any time dependency. It is intended to either plot a single frame of the equation 1, of its steady-state solution.
- `animate_data()`: This method will plot the time-dependent data in an interactive window, such that the user can directly test the quality of their mesh/validity of their data.
- `make_movie()`: This method will output a .mp4 file of the time-dependent data.
- `make_gif()`: This method will output a .gif file of the time-dependent data.

Of most notable note, the interactive widnow generated by `plot_mesh()` will strongly resemble the output as shown in figure 2, but with higher customizability. A sample output is given in figure 7.



(a) Ordered mesh, with quadrangles.

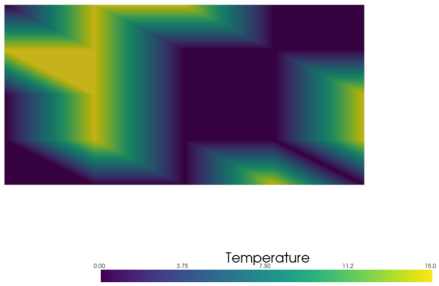


(b) Disordered mesh, with triangles.

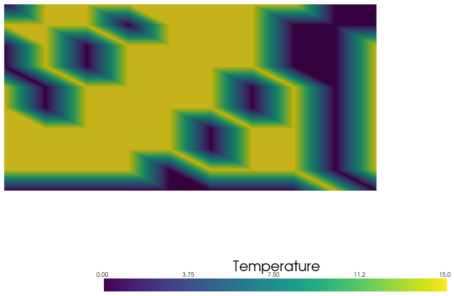
Figure 7: Mesh outputs of `plot_mesh()`, using `PyVista`'s rendering capabilities.

Where the user may also interact with the mesh, and rotate it to any angle.

In addition, a single frame may also be plotted using the `plot_data()` method, given the PDE has already been solved separately, and the content of each node in stored in a 3-dimensional array, for each time-step of the simulation. A sample output is given in figure 8.



(a) Plotted sample data, with 32 elements.



(b) Plotted sample data, with 162 elements.

Figure 8: Outputs of `plot_data()`.

Where the change in the number of elements accurately represents an increase in the resolution and quality of the solution.

In addition, another capability added by the `plotter` branch is the ability to animate the data interactively, and subsequently save a movie or gif of such data. A sample output is given in figure 9.

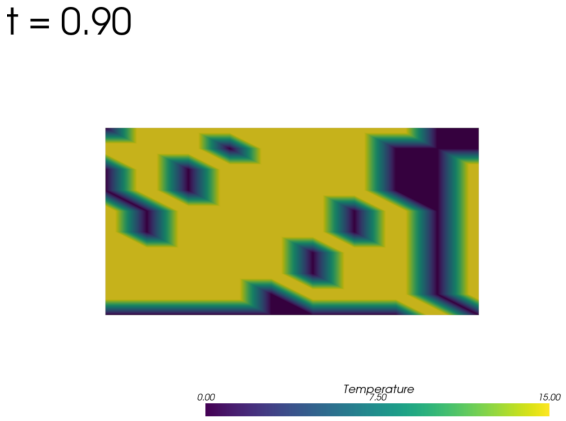


Figure 9: A single frame of the animated data.

Where the most notable difference with the `plot_data()` method is the plotted time at the top-left corner of the figure. In addition, using the `animate_data()` method, the user has control over the number of frames per second of the animation, and enhance readability of the solution. All figures shown above may be generated through a simple test script, as provided in figure 10.

Note: Throughout the development of the `plotter` branch, testing remained a difficult aspect of the project. Many of the methods use rely on visual cues and generated plots to understand their validity. As such, the evaluation of the methods relied primarily in the assertion of the *creation* of the method, rather than its actual content. To achieve this, `PyVista`'s show method (responsible for actually outputting the plot) was mocked using `unittest` and the `MonkeyPatch` testing fixture, such that no plot was actually generated thorugh the tests.

Another particular difficulty lied in the testing of the `make_movie()` and `make_gif()` methods, due to incompabilities of file writing across different operating systems and their reliance on external libraries, which resulted in tests passing locally but not through GitHub's CI pipeline. As such, these tests are currently skipped, while the issue is being actively debugged.

4 Software Quality and Maintenance

`FastFEM` is open-source and version-controlled on GitHub[17]. We are following some rules to ensure high software quality for better maintainability:

- Direct pushes to the "main" branch are not allowed. All the contributions should be made as a pull request.
- Each pull request should be reviewed by two developers, and all the automated tests should be passed before it is merged.
- All the documentation and reports related to the software are maintained in the same repository.
- All the important discussions are conducted on the issues page.
- The changelog is kept.
- Semantic Versioning 2.0.0[23] convention is followed for versioning.

```python
import numpy as np

import fastfem.mesh as m
import fastfem.plotter as p

mesh = m.create_a_rectangle_mesh(
    horizontal_length=2,
    vertical_length=1,
    nodes_in_horizontal_direction=20,
    nodes_in_vertical_direction=20,
    element_type="triangle",
    file_name=None,
)

# Data generation
TOTAL_TIME = 3
FPS = 25
TIME_STEPS = int(TOTAL_TIME * FPS)
DATA = np.random.uniform(
    0,
    50,
    size=(TIME_STEPS, 20, 20),
)

# Visualize
visualizer = p.VisualMesh(mesh)
visualizer.plot_mesh()
visualizer.animate_data(FPS, TOTAL_TIME, DATA)
visualizer.make_movie("filename_movie.mp4", FPS, TOTAL_TIME, DATA)
visualizer.make_gif("filename_gif.gif", FPS, TOTAL_TIME, DATA)
```

Figure 10: Test script demonstrating plotter functions.

`FastFEM` also uses many other open-source tools for various reasons:

- Hatch[24] is used as a project manager.
- MkDocs[25] with Material theme[26] is used for the documentation.
- MkDocstrings[27] is used to generate API references.
- pytest[28] is used to write automated tests.
- GitHub Actions[29] is used to run the automated tests after each update.
- pre-commit[30], Ruff[31], and Black[32] are used for linting and formatting.
- pre-commit.ci is used to run the linters and formatters after each update.

5 Conclusion and Outlook

In conclusion, the current version of `FastFem` is capable of discretizing any PDE into a system of linear equations using the mathematical formalism shown in sections 2 and 3.2, and therefore provide an approximate solution to the PDE given a finite time step, which may be chosen freely by the user at their convenience (and their relative need for accuracy). Coupled with such calculations, `FastFem` also provides a mesh generator using `Gmsh` specified by the domain of the PDE as well as the number of elements required by the user (to be used in conjuction with the elements portion of the code), in addition to a plotter which uses `PyVista` to visualize all results in a 2D space, on the corresponding mesh.

Currently, the full capabilities of `FastFem` are limited to 2D PDEs discretized using triangular elements, which may or may not include time dependency. In the future, we aim to extend `FastFem` to be fully compatible with 3D PDEs and quadrangle elements, as well as introduce the ability to introduce more complex boundary conditions defined by the user. In addition, current plotting methods should enable the user to export their results as a .mp4 or .gif files, but relies on external dependecies that are not necessarily pip-installable, and may therefore not be able to be used by all. In future versions, we aim to make this infrastructure more robust.

References

1. D. Marolf, *Physics Today* **54**, 58–58, ISSN: 0031-9228 (July 2001).
2. M. Feit, J. Fleck, A. Steiger, *Journal of Computational Physics* **47**, 412–433, ISSN: 0021-9991 (1982).
3. T. L. Bergman, *Fundamentals of heat and mass transfer* (John Wiley & Sons, 2011), ISBN: 978-1-119-35388-1.
4. G. Lukaszewicz, P. Kalita, *Navier–stokes equations* (Springer, 2016), vol. 34.
5. G. Barles, H. M. Soner, *Finance and Stochastics* **2**, 369–397 (1998).
6. A. P. Boresi, R. J. Schmidt, *Advanced mechanics of materials* (John Wiley & Sons, 2002), ISBN: 978-1-119-62876-7.
7. D. J. Griffiths, *Introduction to Electrodynamics* (Cambridge University Press, ed. 4, 2017).
8. W. K. Liu, S. Li, H. S. Park, *Archives of Computational Methods in Engineering* **29**, 4431–4453 (2022).
9. P. Dular, C. Geuzaine, *GetDP reference manual: the documentation for GetDP, a general environment for the treatment of discrete problems*, `http://getdp.info`.
10. P. Karban, D. Pánek, T. Orosz, I. Petrášová, I. Doležel, *Computers & Mathematics with Applications* **81**, 618–633 (2021).
11. G. Dhondt, *Munich, Germany, accessed Sept* **21**, 2017, (`http://www.dhondt.de`) (2017).
12. J. Ruokolainen, P. Råback, M. Malinen, T. Zwinger, *ElmerFEM*, (`https://github.com/ElmerCSC/elmerfem`).
13. F. Hecht, *Journal of Numerical Mathematics* **20**, 251–266 (2012).
14. P. R. Schunk *et al.*, (`https://www.osti.gov/biblio/1089869`) (July 2013).
15. I. A. Baratta *et al.*, *DOLFINx: the next generation FEniCS problem solving environment*, preprint, 2023.
16. D. Arndt *et al.*, *Computers & Mathematics with Applications* **81**, 407–422 (Jan. 2021).
17. S. Atalay, S. Escudier, K. Hanson, *FastFEM*, (`http://github.com/fastfem/fastfem`).
18. J. Bradbury *et al.*, *JAX: composable transformations of Python+NumPy programs*, version 0.3.13, 2018, (`http://github.com/jax-ml/jax`).
19. C. B. Sullivan, A. Kaszynski, *Journal of Open Source Software* **4**, 1450, (`https://doi.org/10.21105/joss.01450`) (May 2019).
20. C. Geuzaine, J.-F. Remacle, *International Journal for Numerical Methods in Engineering* **79**, 1309–1331 (2009).
21. A. T. Patera, *Journal of Computational Physics* **54**, 468–488, ISSN: 0021-9991, (`https://www.sciencedirect.com/science/article/pii/0021999184901281`) (1984).
22. A. Quarteroni, R. Sacco, F. Saleri, *Numerical Mathematics* (Springer, 2007), ISBN: 9783540346586, (`https://books.google.com/books?id=Y7grAAAAYAAJ`).
23. T. Preston-Werner, *Semantic Versioning 2.0.0* (`https://semver.org/`).

24. P. P. Authority, *Hatch* (`https://github.com/pypa/hatch`).

25. MkDocs, *MkDocs* (`https://www.mkdocs.org`).

26. M. Donath, *MkDocs-Material* (`https://github.com/squidfunk/mkdocs-material`).

27. MkDocstrings, *MkDocstrings* (`https://github.com/mkdocstrings/mkdocstrings`).

28. H. Krekel *et al.*, *PyTest*, 2004, (`https://github.com/pytest-dev/pytest`).

29. GitHub, *GitHub Actions* (`https://github.com/features/actions`).

30. pre-commit, *pre-commit* (`https://pre-commit.com`).

31. Astral, *Ruff* (`https://github.com/astral-sh/ruff`).

32. Ł. Langa, contributors to Black, *Black: The uncompromising Python code formatter*, (`https://github.com/psf/black`).