XLVI CILAMCE
VITÓRIA - ES - BRAZIL - 2025
XLVI IBERO-LATIN AMERICAN CONGRESS ON COMPUTATIONAL METHODS IN ENGENEERING

Integrating ML-Based Models into the INSANE Finite Element System

Adalberto Horácio de Oliveira Mendes[1], Hugo Mouro Leão[1], Saulo Silvestre de Castro[1] , Roque Luiz da Silva Pitangueira[1]

[1]*Dept. of Structural Engineering, Federal University of Minas Gerais*
Av. Antonio Carlos, 6627, Pampulha, Belo Horizonte, 31270-901, Minas Gerais, Brazil
adalbertohom@gmail.com, hugomleao@yahoo.com.br, saullo9@yahoo.com, roque@dees.ufmg.br

Abstract. Machine learning (ML) models have emerged as promising alternatives to classical constitutive formulations for representing quasi brittle materials, such as concrete, rock, and composite media. Unlike traditional analytical approaches, ML models are trained directly on experimental or simulated data, enabling them to capture complex nonlinear behaviors—such as stiffness degradation, progressive cracking, and brittle-to-ductile transitions—in a more adaptive and generalizable manner. Their inherent flexibility allows for the incorporation of material variability and the customization of constitutive responses according to boundary conditions and loading history. However, the integration of ML-based constitutive models into finite element simulation environments demands substantial modifications to the underlying software architecture. Since such models operate through predictions made by neural networks or similar algorithms, they require dedicated structures for storing and updating internal variables, as well as efficient communication mechanisms with the numerical core to ensure the stability and performance of the analysis. This work presents the strategies adopted to adapt the INSANE system—originally developed in Java—to support ML-based constitutive modeling. Interoperability mechanisms with Python environments were implemented, alongside robust runtime data exchange interfaces and modifications to the constitutive integration logic, while preserving the modularity of the system. The adopted approach enables future extensions to various ML architectures and enhances INSANE's predictive capabilities in simulations involving brittle and heterogeneous materials.

Keywords: Machine Learning, Finite Element Method, Java–Python Interoperability, Quasi-Brittle Materials, Phase-Field

1 Introduction

The advance of machine learning (ML) techniques has driven significant changes in how complex problems are solved numerically [1]. These changes have enabled new approaches to problems for which scientists and engineers have traditionally sought solutions by conventional methods [2] . In constitutive modeling, several proposals have emerged over the years, positioning ML-based constitutive models as promising alternatives to traditional analytical formulations, especially for materials with strong material nonlinearities, such as geotechnical materials, and for quasi-brittle materials, such as concrete, rock, and composites [3–5]. These models are trained directly on experimental or simulated data, which allows them to capture highly nonlinear behaviors, including stiffness degradation and progressive cracking [3].

Embedding ML-based constitutive models into legacy finite element method (FEM) simulation environments poses considerable challenges, particularly with respect to the architecture of existing computational systems. ML models operate through inferences performed by neural networks or similar algorithms, requiring dedicated structures for storing internal variables, integration logic, and efficient communication mechanisms between the numerical kernel and the inference modules. Overcoming these challenges is essential to ensure the stability, accuracy, and computational performance of the simulations. Another important aspect concerns the compatibility between the original language of the legacy system and the ML models. Many ML models are developed in Python due to its wide applicability in data science; however, finite element applications commonly developed on lower-level languages such as C++, Java, and, in older systems, Fortran.

This study presents the solutions implemented in the INSANE system, a numerical simulation platform originally developed in Java, to enable support for ML-based constitutive models. To this end, interoperability mechanisms with Python environments, interfaces for run-time data exchange, and modifications to the constitutive

integration logic were implemented, while preserving the modularity and extensibility of the original system. The proposed approach is designed to allow future extensions to different learning architectures and to enhance INSANE's predictive capabilities in the simulation of brittle and heterogeneous materials.

2 General Coupling Strategy with Python

Integrating ML models with the INSANE system required developing an interoperability mechanism between the numerical core, implemented in Java, and the prediction modules implemented in Python. The adopted strategy was guided by three fundamental requirements: (i) low intrusiveness in the legacy code to preserve system modularity; (ii) runtime data exchange with minimal computational overhead; and (iii) support for future extensions to different ML architectures.

To enable the coupling, an embedded Python interpreter was used, accessed directly from the Java environment through the JEP (Java Embedded Python) library [6]. This approach was selected after evaluating several available alternatives.

Unlike options such as Py4J, REST APIs, or gRPC—which require running the Python interpreter as an external process and exchanging data via interprocess communication (IPC)—JEP allows a CPython interpreter to run inside the JVM, in the same Java process, via JNI. This eliminates the need for (de)serialization of data in most cases, significantly reducing call latency and computational overhead—factors that are critical when many constitutive-model invocations occur per load increment.

Alternatives such as Jython and JPype, in turn, pose important limitations. Jython is incompatible with libraries commonly used in machine learning workflows, such as NumPy, SciPy, or TensorFlow, because it does not implement the CPython API. JPype, while enabling Java–Python calls via JNI, was primarily designed to execute Java code from a Python interpreter and is not optimized for the inverse flow—i.e., running Python models from a Java-centered environment.

Given these limitations, JEP stands out by combining in-process execution with low latency, full compatibility with the Python scientific ecosystem, and direct control of interpreter sessions from Java, enabling efficient management of multiple interpreters in multi-threaded contexts. Moreover, its integration is non-intrusive, preserving INSANE's modular architecture and allowing ML models to be replaced or expanded without structural changes to the system core.

3 Python Interfacing Layer

To ensure efficient integration between INSANE and machine-learning models developed in Python, an intermediate layer called the Python Interfacing Layer (PIL) was created. This layer is designed to abstract the operational details of Java–Python communication, providing a high-level interface for model execution, variable management, and runtime data exchange.

The PIL is built on three main components: the `Python Session Manager`, `Java Python variable binders`, and `ML Predictor Engine`. Each of these components is described in detail in the following sections.

3.1 Python Session Manager

The `PythonSessionManager` class was developed to manage the consistent and safe use of the embedded Python interpreter in INSANE, in accordance with the constraints imposed by the JEP library. This library requires that each instance of `SharedInterpreter` be created and used exclusively by the same thread throughout its lifetime.

The primary goal of `PythonSessionManager` is to ensure that each thread requesting access to the Python environment always receives the same interpreter instance, even when the requests occur at different times or in different parts of the code. To achieve this, the class uses a `ThreadLocal<SharedInterpreter>` structure that transparently binds a unique interpreter instance to each calling thread. This instance is created on demand—at first use—and remains valid until it is explicitly closed.

The class interface exposes simple methods for acquiring and releasing the interpreter, encapsulating the instance lifecycle and ensuring that Python function execution complies with JEP's usage model. This prevents context failures, concurrency errors, and undefined behavior, even when multiple consecutive calls to the interpreter occur within the same thread.

By ensuring that each thread operates with a single, reusable, and isolated interpreter instance, this solution enables safe integration between Java numerical modules and Python inference models, while preserving

INSANE's modularity, stability, and predictability.

3.2 Java–Python Variable Binders

A central component of the Python Interfacing Layer (PIL) is the mechanism that enables synchronized data exchange between Java objects and Python variables. This mechanism is implemented through classes called binders, which abstract the creation, update, and management of variables across the Java and Python environments. All binders derive from a generic abstract class named PythonVariableBinder¡T¿, where T denotes the Java data type being mapped.

Each binder instance is responsible for establishing and maintaining the correspondence between a Java object and a variable in the Python interpreter's namespace. At creation time, the binder allocates the Python variable under a specific name and registers it in a centralized namespace manager, preventing name collisions. During execution, the update() method is used to push new values to the Python environment whenever the underlying Java object changes. This enables real-time synchronization, which is important in simulations that frequently update input and output variables across both languages.

The binder design fully abstracts low-level interoperability operations, allowing developers to focus on numerical logic rather than encoding or serialization details. It also ensures consistent and safe memory usage by providing controlled lifecycle management.

Several specialized implementations were developed to cover different data types. TablesawTableToDataFrameBinder converts Java Tablesaw Table objects into Python pandas.DataFrame structures, facilitating the structured transfer of tabular input data. NumpyArrayToJavaArrayBinder transfers NumPy arrays produced in Python to native Java arrays, which is necessary for receiving model outputs such as stress predictions or updated internal variables. A more generic class, HashMapToDataFrameConverterBinder, was created to map HashMap¡String, Object¿ instances to Python dictionaries or DataFrames, enabling the exchange of metadata or custom state representations.

The binder mechanism provides several benefits: it encapsulates all transformation and data-transfer logic into reusable components, promotes type safety, and simplifies future extensions to new data structures. A key motivation for developing this mechanism was to preserve the PIL's modularity and extensibility, enabling hybrid simulation workflows in which computational mechanics routines remain in Java while machine learning models run natively in Python.

3.3 ML Predictor Engine

ML prediction executors constitute an abstraction within the Python Interfacing Layer (PIL), enabling transparent invocation of machine-learning models implemented in Python from the INSANE system. These classes encapsulate all logic required to prepare input data, trigger prediction, and retrieve results, ensuring that communication between the two environments remains modular, maintainable, and compatible with the computational demands of FEM simulations.

Each instance of MLPredictorEngine manages the full lifecycle of a prediction call in the Python environment. Internally, it holds references to one or more input binders that synchronize the current state of Java variables (e.g., strain tensors, internal variables) with their corresponding representations in the Python interpreter. Likewise, the executor maintains output binders used to extract inference results—typically stress tensors or updated internal variables—back into the Java environment.

Inference is typically orchestrated by the predict() method, which executes the following steps in sequence:

Input binder update. All input binders are refreshed to transfer the most recent Java-side values to Python before inference starts.

Interpreter acquisition:.an instance of the Python interpreter is obtained from the PythonSessionManager.

nference function call. the Python function responsible for prediction is invoked via interpreter.exec(), depending on whether a return value is required. The function is designed to operate on variables already present in the interpreter's namespace.

Output synchronizationAfter execution, the output binders retrieve the inference results and update the corresponding Java variables. This step may include data transformation, such as converting NumPy arrays into native Java arrays.

The implemented classes emphasize flexibility and compatibility with different ML model architectures and data formats. They serve as a bridge between INSANE's numerical core and Python-based models, exposing a simple interface while fully abstracting cross-language communication and interpreter management.

This architectural separation is essential for deploying machine-learning models in real simulation workflows. It allows models to be developed directly in Python, using domain-specific frameworks, and subsequently integrated into INSANE without changes to the numerical core.

4 Application Example: Phase-Field Simulation

As a direct application of the integration mechanisms developed in this work, a modified version of the `convergePhaseField()` method from the `PhaseFieldStandardNewtonRaphsonStaggeredSolver` class was implemented, aiming to **reduce the computational cost** associated with the update of the damage field (phase field) in phase-field simulations. Traditionally, this update requires solving a coupled partial differential equation using iterative numerical methods, which involves multiple cycles of system assembly and solution. In the proposed extension, this stage is replaced by a **runtime inference** step using a machine learning (ML) model previously trained on reference simulations.

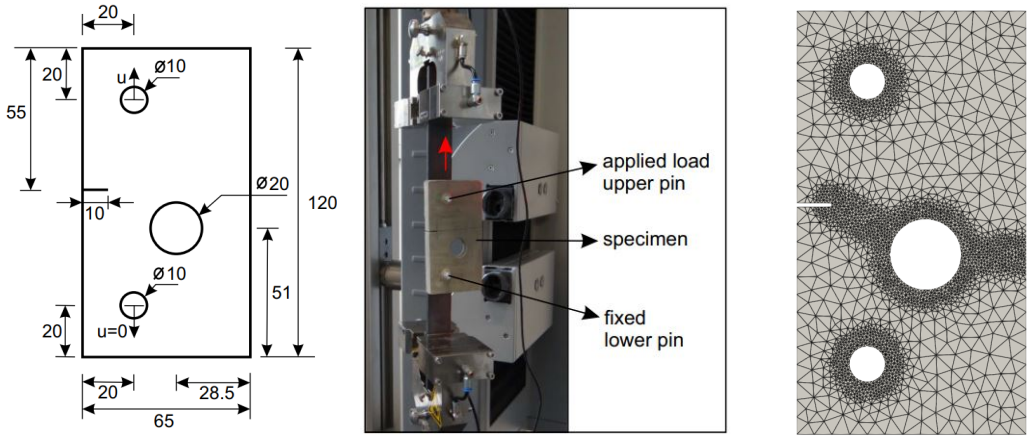
At each iteration of the solver, the current nodal strain data are extrapolated from the Gauss points and structured into a `Tablesaw` table. This table is then transferred to the embedded Python interpreter using the `TablesawTableToDataFrameBinder`, which converts the table into a `pandas.DataFrame` compatible with the ML model.

The inference process is performed by the `MLPredictorEngine`, which loads a serialized Python pipeline and executes the prediction step within the same Java process. The predicted damage field, returned as a NumPy array, is retrieved in Java via the `NumpyToJavaArrayBinder`, and converted into an `IVector` structure compatible with INSANE's internal data representation.

The embedded interpreter is managed by the `PythonSessionManager`, which ensures that each Java thread consistently reuses the same `SharedInterpreter` instance throughout its execution. This strategy complies with the constraints of the JEP library and avoids concurrency issues or redundant interpreter creation.

The routine also records the execution time of each iteration in CSV files for performance diagnostics.

To verify the functionality of the proposed coupling architecture, a numerical model was executed to reproduce the experimental setup presented by [7]. The test consists of a notched cementitious plate subjected to loading through top and bottom pins, featuring an off-centered hole that induces mixed-mode fracture. The specimen geometry, boundary conditions, and experimental configuration were kept identical to the original study, as illustrated in Figures 1a and 1b.



(a) (b) (c)

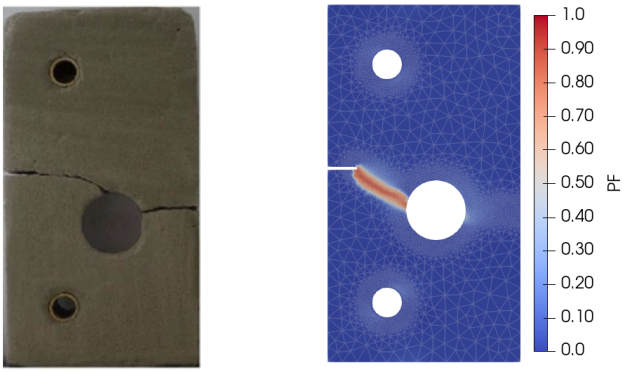
Figure 1. (a) Test dimensions notched plate with hole; (b) Test notched plate with hole; (c) Mesh Plate notched with hole.

The numerical model using the previously described extension, which incorporates the prediction of the phase field through a machine learning (ML) model. The mesh consists of 4,550 triangular elements (1c), refined in regions of higher interest, particularly near the notch and the hole.

Simulations were conducted using fixed displacement increments of mm per step. $\Delta u = 1 \times 10^{-4}$ It is important to note that the dataset used for training the ML model was generated with the same constitutive prop-

erties applied in the validation simulation, ensuring compatibility between the model's generalization domain and the analyzed problem. Therefore, the simulation represents results obtained for a material with the following properties: Young's modulus $E = 21876.0kN/mm^2$, Poisson's ratio $\nu = 0.18$, critical energy release rate $G_c = 1.8 \times 10^{-2}kN/mm$, and length scale parameter $l_0 = 2.234mm$.

The computational results of validation test are presented in Figure 2.



(a) (b)

Figure 2. (a) Experimental observation of fractured specimen; (b) Predicted crack path by MEF method

By comparing the experimental results (Figure2a) with the degradation pattern observed in the numerical simulation (Figure2b), it becomes evident that the predicted behavior is qualitatively consistent with the physical response observed in the laboratory. This agreement indicates that the coupling architecture and the implemented extensions were correctly integrated into the simulation environment, demonstrating that the proposed solution is functional and fit for purpose.

It is important to emphasize that the quality of the numerical response depends directly on the accuracy of the machine learning model used to infer the damage field. However, the evaluation of the predictive performance of ML models falls outside the scope of this work and will be addressed in future studies.

Therefore, it can be concluded that the integration strategies presented in this article—including the Python session manager, the Java–Python variable binders, and the inference executors—are appropriate for enabling the efficient use of ML-based constitutive models within the INSANE simulation framework.

5 Conclusions

This work presented a modular and non-intrusive architecture for integrating ML-based constitutive models into the INSANE finite element system. The proposed solution relies on a dedicated Python Interfacing Layer, which ensures safe interpreter management, efficient runtime data exchange, and seamless communication between Java and Python.

The effectiveness of the implementation was demonstrated through a phase-field simulation, in which the damage field was predicted via a trained ML model. The results showed qualitative agreement with experimental data, confirming the viability of the proposed coupling strategy.

While the accuracy of the ML model remains a key factor—subject to future investigation—the developed architecture proved robust and extensible, enabling the use of machine learning in high-performance finite element simulations.

Acknowledgements. The authors gratefully acknowledge the support from the Brazilian research agencies CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior), FAPEMIG (Fundação de Amparo à Pesquisa do Estado de Minas Gerais; Grant PPM-00747-18), and CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico; Grant 309515/2017-3).

Authorship statement. The authors hereby confirm that they are the sole liable persons responsible for the authorship of this work, and that all material that has been herein included as part of the present paper is either the property (and authorship) of the authors, or has the permission of the owners to be included here.

References

[1] T. M. Mitchell and others. Machine learning, 1997.
[2] 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*, vol. 378, pp. 686–707, 2019.
[3] S. Rezaei, A. Moeineddin, and A. Harandi. Learning solutions of thermodynamics-based nonlinear constitutive material models using physics-informed neural networks. *Computational Mechanics*, vol. 74, n. 2, pp. 333–366, 2024.
[4] A. F. Figueiredo. The chalenges of concrete constitutive modeling via artificial neural networks. *Federal Univesity of Minas Gerais (UFMG)*, vol. , 2023.
[5] A. Hussain, A. H. Sakhaei, and M. Shafiee. Machine learning-based constitutive modelling for material nonlinearity: A review. *Mechanics of Advanced Materials and Structures*, pp. 1–19, 2024.
[6] JEP Developers. Jep - java embedded python. `https://github.com/ninia/jep`. Accessed: July 2025, 2025.
[7] M. Ambati, T. Gerasimov, and L. De Lorenzis. A review on phase-field models of brittle fracture and a new fast hybrid formulation. *Computational Mechanics*, vol. 55, pp. 383–405, 2015.