FEA Surrogate Model

Objective

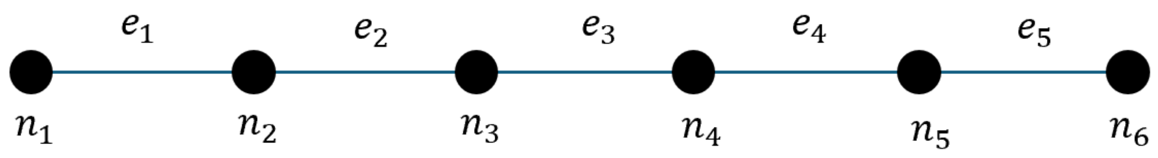
The goal of this project is to outline an overview of the process of feature extraction from the discretization of Finite Element Analysis (FEA) Models. The end goal is to develop a proof of concept machine learning model that proves the predictive capability for surrogate models to predict the same outputs as an FEA model.

Overview

The Finite Element Method uses a discretization technique that creates finite elements in a 1-, 2-, and 3-dimensional geometry. Engineers use this method to determine localized information to design thermal, mechanical, and electrical systems. The method presents unique challenges to ensure the results of the analyses are accurate and provide quality insight into the physical behavior of a system.

Basics of FEA

To highlight the basics of FEA, the following 1-dimensional, 6-node, and 5-element model will be the baseline



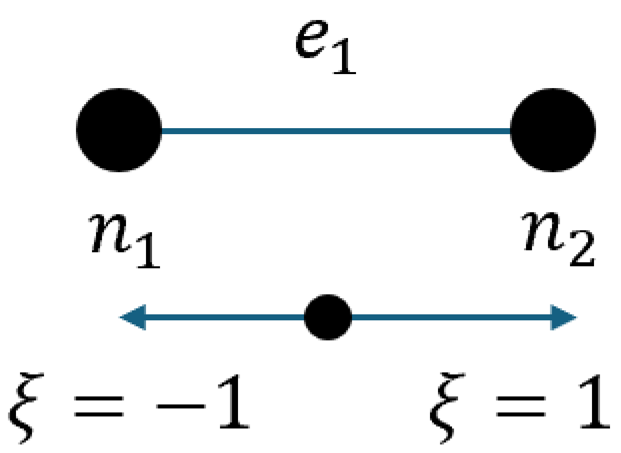
As outlined by Cert, the goal of the FEA model is to develop a system of equations that follow the form [1]:

$$[K][T] = [F] + [B]$$

Where [K] and [F] can be defined as follows:

$$[K] = \sum_{e}^{NE} [K^e] \; and \; [F] = \sum_{e=1}^{NE} [F^e]$$

Defining the local coordinate system, which applies to each element individually:



Where the shape functions for the first element are defined as follows:

$$S_1^1 = \frac{1}{2}(1-\xi) \; and \; S_2^1 = \frac{1}{2}(1+\xi)$$

For the following form of differential equation:

$$u\frac{dT}{dx} - k\frac{d^2T}{dx^2} = f \; in \; 0 \leq x \leq 1$$

Where u is the convective heat transfer coefficient, k is the conductive heat transfer coefficient, and f is the volumetric heat generation.

$$u = \frac{W}{m^2K}, \; k = \frac{W}{mK}, \; f = \frac{W}{m^3}$$

The local shape function can be calculated element by element with the following equation:

$$K_{ij}^e = \int_{-1}^{1} \left(S_i u \frac{dS_j}{d\xi} \frac{1}{J^e} + k \frac{dS_i}{d\xi} \frac{1}{J^e} \frac{dS_j}{d\xi} \frac{1}{J^e} \right) J^e d\xi$$

Where the Jacobian can be defined as follows:

$$J^e = \frac{dx}{d\xi} = \frac{h^e}{2}$$

Let's assume that $h^e = 1$ for each element. We can then write the Jacobian, for each element, as follows:

$$J^e = \frac{1}{2}$$

Finally, we define the internal heat general equation as follows (2.26):

$$F_i^e = \int_{-1}^{1} S_i^e f J^e d\xi$$

Now, to define the equations for the first element.

$$K_{11}^1 = \int_{-1}^{1} \left(S_1^1 u \frac{dS_1^1}{d\xi} \frac{1}{J^e} + k \frac{dS_1^1}{d\xi} \frac{1}{J^e} \frac{dS_1^1}{d\xi} \frac{1}{J^e} \right) J^e d\xi$$

$$K_{12}^1 = \int_{-1}^{1} \left(S_2^1 u \frac{dS_2^1}{d\xi} \frac{1}{J^e} + k \frac{dS_1^1}{d\xi} \frac{1}{J^e} \frac{dS_2^1}{d\xi} \frac{1}{J^e} \right) J^e d\xi$$

$$K_{21}^1 = \int_{-1}^{1} \left(S_2^1 u \frac{dS_1^1}{d\xi} \frac{1}{J^e} + k \frac{dS_2^1}{d\xi} \frac{1}{J^e} \frac{dS_1^1}{d\xi} \frac{1}{J^e} \right) J^e d\xi$$

$$K_{22}^1 = \int_{-1}^{1} \left(S_2^1 u \frac{dS_2^1}{d\xi} \frac{1}{J^e} + k \frac{dS_2^1}{d\xi} \frac{1}{J^e} \frac{dS_2^1}{d\xi} \frac{1}{J^e} \right) J^e d\xi$$

$$F_1^1 = \int_{-1}^{1} S_1^1 f J^1 d\xi$$

$$F_2^1 = \int_{-1}^{1} S_2^1 f J^1 d\xi$$

Now, we will determine the local stiffness matrix for the first element, and the internal heat generation.

$$K_{11}^1 = \int_{-1}^{1} \left(S_1^1 u \frac{dS_1^1}{d\xi} \frac{1}{J^e} + k \frac{dS_1^1}{d\xi} \frac{1}{J^e} \frac{dS_1^1}{d\xi} \frac{1}{J^e} \right) J^e d\xi$$

$$= \int_{-1}^{1} \left[\left(\frac{1}{2} (1 - \xi) \left(\frac{-1}{2} \right) 2u \right) + k \left(\frac{1}{4} 4 \right) \right] \left(\frac{1}{2} \right) d\xi$$

$$= \int_{-1}^{1} \frac{1}{4} (-1 + \xi) u + \left(\frac{1}{2} \right) k d\xi$$

$$= \left[\frac{1}{4} \left(-\xi + \frac{\xi^2}{2} \right) u + k \left(\frac{\xi}{2} \right) \right]_{-1}^{1}$$

$$= \left[\frac{1}{4} \left(-1 + \frac{1}{2} \right) u + \frac{k}{2} \right] - \left[\frac{1}{4} \left(1 - \frac{1}{2} \right) u - \frac{k}{2} \right]$$

$$= \left[\frac{1}{4} \left(\frac{-1}{2} \right) u + \frac{k}{2} \right] - \left[\frac{1}{4} \left(\frac{1}{2} \right) u - \frac{k}{2} \right]$$

$$= \left[\frac{-u}{8} + \frac{k}{2} - \frac{u}{8} + \frac{k}{2} \right]$$

$$= \left[\frac{-1}{4} u + k \right] = K_{11}^1$$

Now, if we assume that there is no convection, then we can say u = 0 and write the following:

$$K_{11}^1 = k$$

For the second element in the local stiffness matrix, assuming u = 0 to neglect convection:

$$K_{12}^1 = \int_{-1}^{1} \left(k \frac{dS_1^1}{d\xi} \frac{1}{J^e} \frac{dS_2^1}{d\xi} \frac{1}{J^e} \right) J^e d\xi$$

$$= \int_{-1}^{1} \left(k \left(-\frac{1}{2} \right) \left(\frac{-1}{2} \right) (2)^2 \right) \frac{1}{2} d\xi$$

$$= \int_{-1}^{1} - \left(\frac{k}{2} \right) d\xi$$

$$= \left[-\frac{k\xi}{2} \right]_{-1}^{1} = \left[\frac{-k}{2} (1 - (-1)) \right] = \left[\frac{-k}{2} (2) \right] = -k = K_{12}^1$$

For the third element in the local stiffness matrix, assuming u = 0 to neglect convection:

$$K_{21}^1 = \int_{-1}^{1} \left(k \frac{dS_2^1}{d\xi} \frac{1}{J^e} \frac{dS_1^1}{d\xi} \frac{1}{J^e} \right) J^e d\xi$$

$$= \int_{-1}^{1} \left(k \left(-\frac{1}{2} \right) \left(\frac{-1}{2} \right) (2)^2 \right) \frac{1}{2} d\xi$$

$$= \int_{-1}^{1} - \left(\frac{k}{2} \right) d\xi$$

$$= \left[-\frac{k\xi}{2} \right]_{-1}^{1} = \left[\frac{-k}{2} (1 - (-1)) \right] = \left[\frac{-k}{2} (2) \right] = -k = K_{21}^1$$

For the fourth element in the local stiffness matrix, assuming u=0 to neglect convection:

$$K_{22}^1 = \int_{-1}^{1} \left(k \frac{dS_2^1}{d\xi} \frac{1}{J^e} \frac{dS_2^1}{d\xi} \frac{1}{J^e} \right) J^e d\xi$$

$$= \int_{-1}^{1} \left(k \left(-\frac{1}{2} \right)^2 (2)^2 \right) \frac{1}{2} d\xi$$

$$= \int_{-1}^{1} \left(\frac{k}{2} \right) d\xi$$

$$= \left[\frac{k\xi}{2} \right]_{-1}^{1} = \left[\frac{k}{2}(1-(-1)) \right] = \left[\frac{k}{2}(2) \right] = k = K_{22}^{1}$$

Now, assembling the local stiffness matrix for element 1:

$$K^1 = \begin{bmatrix} k & -k \\ -k & k \end{bmatrix}$$

Since each element is the same size and has the same shape function, we can say the following:

$$K^1 = K^2 = K^3 = K^4 = \begin{bmatrix} k & -k \\ -k & k \end{bmatrix}$$

Now, assembling the global stiffness matrix:

$$K = \begin{bmatrix} k & -k & 0 & 0 & 0 & 0 \\ -k & (k+k) & -k & 0 & 0 & 0 \\ 0 & -k & (k+k) & -k & 0 & 0 \\ 0 & 0 & -k & (k+k) & -k & 0 \\ 0 & 0 & 0 & -k & (k+k) & -k \\ 0 & 0 & 0 & 0 & -k & k \end{bmatrix}$$

Simplifying as follows:

$$K = k \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix}$$

Now, defining the [T] vector as the unknown temperatures of the nodes in the system. With no boundary conditions, the vector can be defined as follows:

$$[T] = \begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \\ T_5 \\ T_6 \end{bmatrix}$$

For the first element, assuming f is not dependent on ξ:

$$F_1^1 = \int_{-1}^{1} S_1^1 f J^1 d\xi = \int_{-1}^{1} \left(\frac{1}{2}(1-\xi)\right) f \left(\frac{1}{2}\right) d\xi = \frac{f}{4}\int_{-1}^{1} (1-\xi)\, d\xi$$

$$= \frac{1}{4}\left[\xi - \frac{\xi^2}{2}\right]_{-1}^{1} = \frac{1}{4}\left[1-\frac{1}{2}\right] - \frac{1}{4}\left[-1-\frac{1}{2}\right] = \frac{1}{4}[2] = \frac{f}{2}$$

For the second element, assuming f is not dependent on ξ:

$$F_2^1 = \int_{-1}^{1} S_2^1 f J^1 d\xi = \int_{-1}^{1} \left(\frac{1}{2}(1+\xi)\right) f \left(\frac{1}{2}\right) d\xi = \frac{f}{4}\int_{-1}^{1} (1+\xi) d\xi$$

$$= \frac{1}{4}\left[\xi + \frac{\xi^2}{2}\right]_{-1}^{1} = \frac{f}{4}\left[1+\frac{1}{2}\right] - \frac{f}{4}\left[-1+\frac{1}{2}\right] = \frac{f}{4}[2] = \frac{f}{2}$$

Therefore, for each element, the local force vector is as follows:

$$F^1 = F^2 = F^3 = F^4 = F^5 = \begin{bmatrix} \frac{f}{2} \\ \frac{f}{2} \end{bmatrix} = f\begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \end{bmatrix}$$

Now, configuring the global force vector:

$$F = f\begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} + \frac{1}{2} \\ \frac{1}{2} + \frac{1}{2} \\ \frac{1}{2} + \frac{1}{2} \\ \frac{1}{2} + \frac{1}{2} \\ \frac{1}{2} \end{bmatrix} = f\begin{bmatrix} \frac{1}{2} \\ 1 \\ 1 \\ 1 \\ 1 \\ \frac{1}{2} \end{bmatrix}$$

Which results in the following system:

$$k\begin{bmatrix}1 & -1 & 0 & 0 & 0 & 0\\ -1 & 2 & -1 & 0 & 0 & 0\\ 0 & -1 & 2 & -1 & 0 & 0\\ 0 & 0 & -1 & 2 & -1 & 0\\ 0 & 0 & 0 & -1 & 2 & -1\\ 0 & 0 & 0 & 0 & -1 & 1\end{bmatrix}\begin{bmatrix}T_1\\ T_2\\ T_3\\ T_4\\ T_5\\ T_6\end{bmatrix} = \begin{bmatrix}\frac{f}{2}\\ f\\ f\\ f\\ f\\ \frac{f}{2}\end{bmatrix}$$

The vector above is only valid for values or functions of f that are not dependent on ξ. Now, the boundary condition vector must be defined. As outlined by Cert, the three types of boundary conditions are essential, natural or mixed [1]. To demonstrate an essential boundary condition, it is assumed that the temperature at node 6, T_6, is known. This results in the following system:

$$k\begin{bmatrix}1 & -1 & 0 & 0 & 0\\ -1 & 2 & -1 & 0 & 0\\ 0 & -1 & 2 & -1 & 0\\ 0 & 0 & -1 & 2 & -1\\ 0 & 0 & 0 & -1 & 2\end{bmatrix}\begin{bmatrix}T_1\\ T_2\\ T_3\\ T_4\\ T_5\end{bmatrix} = \begin{bmatrix}\frac{f}{2}\\ f\\ f\\ f + kT_6\end{bmatrix}$$

Now, a natural boundary condition can be resembled by knowing a value B_1, a metric at node 1. Implementing into our system of equations as follows:

$$k\begin{bmatrix}1 & -1 & 0 & 0 & 0\\ -1 & 2 & -1 & 0 & 0\\ 0 & -1 & 2 & -1 & 0\\ 0 & 0 & -1 & 2 & -1\\ 0 & 0 & 0 & -1 & 2\end{bmatrix}\begin{bmatrix}T_1\\ T_2\\ T_3\\ T_4\\ T_5\end{bmatrix} = \begin{bmatrix}\frac{f}{2}\\ f\\ f\\ f\\ f + kT_6\end{bmatrix} + \begin{bmatrix}B_1\\ 0\\ 0\\ 0\\ 0\end{bmatrix}$$

Now, for a mixed type boundary condition:

$$k\begin{bmatrix}1 & -1 & 0 & 0 & 0\\ -1 & 2 & -1 & 0 & 0\\ 0 & -1 & 2 & -1 & 0\\ 0 & 0 & -1 & 2 & -1\\ 0 & 0 & 0 & -1 & 2\end{bmatrix}\begin{bmatrix}T_1\\ T_2\\ T_3\\ T_4\\ T_5\end{bmatrix} = \begin{bmatrix}\frac{f}{2}\\ f\\ f\\ f + kT_6\end{bmatrix} + \begin{bmatrix}\alpha T_1 + \beta\\ 0\\ 0\\ 0\\ 0\end{bmatrix}$$

Where the system can be simplified as follows:

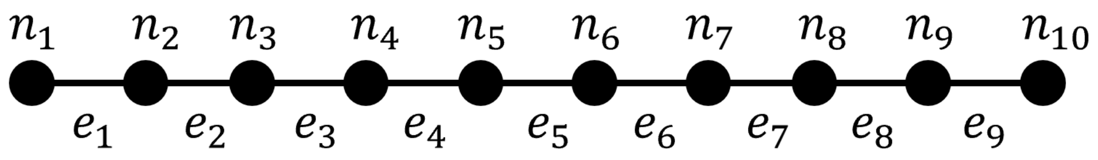
$$k\begin{bmatrix}1-\alpha & -1 & 0 & 0 & 0\\ -1 & 2 & -1 & 0 & 0\\ 0 & -1 & 2 & -1 & 0\\ 0 & 0 & -1 & 2 & -1\\ 0 & 0 & 0 & -1 & 2\end{bmatrix}\begin{bmatrix}T_1\\ T_2\\ T_3\\ T_4\\ T_5\end{bmatrix} = \begin{bmatrix}\frac{f}{2}\\ f\\ f\\ f + kT_6\end{bmatrix} + \begin{bmatrix}\beta\\ 0\\ 0\\ 0\\ 0\end{bmatrix}$$

The basics of FEA modelling have been outlined. A mesh was created for a simple 6-node system, and the system of equations were generated, for varying boundary conditions.

Batch FEA Model for Data Generation

In this section, the basics of FEA will be used to generate a plethora of data to train and test the Surrogate Machine Learning Model. The goal is to generate the required data to train a machine learning (ML) model.

The number of nodes and elements are increased to 10 and 9, respectively. The updated model can be defined as follows:



The next step requires the generation of data with an exhaustive approach. The method uses four nested loops that iterate over parameters to compile various combinations that affect the FEA model. The exhaustive approach ensures that the dataset is comprehensive for the ML model.

The first nested loop varies the stiffness inside of the stiffness matrix. The global stiffness matrix for this model as follows, prior to any boundary conditions being applied:

$$K_{Global} = k \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix}$$

The stiffness, k, is the parameter varied in the first nested loop. The parameter has 100 quantities ranging from 5 to 200.

The second nested loop iterates through the different nodes in the model. For each stiffness value, the boundary conditions and external quantities are applied to specific nodes. The location refers to the node which a boundary condition is applied (whether it's essential, natural, or mixed).

The third nested loop applies the various boundary conditions to the system. Boundary conditions dictate how the system behaves at its edge or points of support. The conditions vary from essential boundary conditions (Dirichlet), where the temperature is fixed, to natural boundary conditions (Neumann), where external forces are applied, to mixed conditions, which are combinations of the two.

The fourth nest loop iterates over a range of values for the external force or load applied at each node. The quantity influences the system by impacting the response of the system. The range for the quantity could span from 1 to 150, with values chosen to simulate a range of loading conditions.

The four combined nested loops generate 422,000 data points which will be by iterating through stiffnesses, boundary conditions, locations of the boundary conditions, and the magnitude of force applied at each node.

Machine Learning Model

The Machine Learning Model trained both a Neural Network and Gradient Boosting Model. These models were chosen for their ability to capture complex relationships between features and predict continuous values, such as the temperature distributions from the FEA model. The data for both models was split using an 80/20 ratio (80% for training, 20% for testing) and the regressors were scaled using StandardScaler to accommodate the wide range of values in the features and targets.

Gradient Boosting Model

The Gradient Boosting Model (GBM) is another powerful regression technique used for predicting continuous outcomes. The Gradient Boosting Model builds an ensemble of weak learners (usually decision trees) that are trained sequentially.

Hyperparameter Tuning - Best Parameters

Hyperparameter	Best
Hidden Layer Sizes	50
Activation	tanh
Solver	adam
Alpha	0.01
Learning Rate	constant
Initial Learning Rate	0.001
Maximum Iterations	5000
Batch Size	32

Results

Dataset	Mean Square Error (MSE)	R^{2}‘
Training	0.0013	0.8601
Testing	0.0013	0.8594

The gradient boosting model showed a stable performance with both the training and testing data, with an R^2 score of approximately 0.86, indicating that the model is able to explain a significant portion of the variance in the temperature distributions.

Neural Network

The Neural Network model used a Multi-layer Perceptron (MLP) regressor, which can handle regression tasks where the goal is to predict continuous values. The MLP regressor is a type of feedforward neural network that consists of an input layer, one or more hidden layers, and an output layer. It is particularly effective when dealing with non-linear relationships between inputs and outputs.

Hyperparameter Tuning

Hyperparameter	Best
Number of Estimators	100
Learning Rate	0.01
Max Depth	5
Minimum Samples Split	2
Minimum Samples Leaf	1
Subsample	0.8

Results

Dataset	Mean Square Error (MSE)	R^{2}‘
Training	0.0018	0.9981
Test	0.0019	0.9982

The Neural Network produced an R^2 score of 0.9981 on the training set and 0.9982 on the test set, indicating very accurate predictions. However, this may be indicative of overfitting.

Comparison of Models

The Neural Network model showed excellent results with an R^2 close to 1 on both training and testing datasets, indicating the models was effective at learning learning the relationships between the features and target temperatures, with there being a chance for overfitting.

The Gradient Boosting model had slightly lower performance, but still managed to capture the main trends with an R^2 of 0.86.

Next Steps and Potential Improvements

Both models were able to predict the temperature distributions effectively, with the Neural Network outperforming the Gradient Boosting model in terms of accuracy. Moving forward, the models can be refined by incorporating more features, performing feature extraction, or fine-tune hyperparameters further to improve performance.

Alternatively, the Neural Network should be be trained with more data to ensure that it isn't overfitting to the existing data.

Conclusion

In this project, surrogate models were developed to approximate Finite Element Analysis (FEA) simulations. By generating a comprehensive dataset through exhaustive simulations, Neural Network and Gradient Boosting models were trained and testes to predict temperature distributions across various structural configurations. The results demonstrated that these surrogate models could effectively replicate FEA outcomes, significantly reducing computational time while maintaining accuracy.

Works Cited

[1] *Dr. Cuneyt Sert's blog*. (n.d.). https://users.metu.edu.tr/csert/teaching_notes.htm

Appendix

Neural Network Code

```
import numpy as np
import itertools
```

```
from sklearn.model_selection import GridSearchCV
from sklearn.neural_network import MLPRegressor
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
import pandas as pd

# ==============================
# FEA Batch Model
# ==============================
class OneD:
    def __init__(self, Elements, Nodes, k):
        self.Elements = Elements
        self.Nodes = Nodes
        self.K_val = k
        self.K_matrix = np.zeros((self.Nodes, self.Nodes))
        self.Local_k = np.array([[1, -1], [-1, 1]])
        self.Force = np.zeros((self.Nodes, 1))

    def K_stiffness(self):
        for j in range(0, self.Nodes - 1):
            self.K_matrix[j:j+2, j:j+2] += self.K_val * self.Local_k
        return self.K_matrix

    def apply_boundary_condition(self, BC, Location, Quantity):
        if BC == 0:  # Essential (Dirichlet)
            self.Force[Location] += self.K_matrix[Location, Location] * Quantity
            self.K_matrix = np.delete(self.K_matrix, Location, axis=0)
            self.K_matrix = np.delete(self.K_matrix, Location, axis=1)
            self.Force = np.delete(self.Force, Location, axis=0)

        elif BC == 1:  # Natural (Neumann)
            self.Force[Location] += Quantity

        elif BC == 2:  # Mixed
            self.K_matrix[Location, Location] += self.K_val
            self.Force[Location] += self.K_val * Quantity
    def solve(self):
        try:
            T = np.linalg.solve(self.K_matrix, self.Force)
            return T
        except np.linalg.LinAlgError:
            return None

# Exhaustive generation parameters
Nodes = 10
Elements = Nodes - 1
k_values = np.linspace(5, 200, 100)  # Vary stiffness from 5 to 200
BC_types = [0, 1, 2]
Quantity_range = np.linspace(1, 150, 200)  #  values from 1 to 150

# Store results
dataset = []

# Loop through k values, all nodes, BC types and quantities
for k in k_values:
    for node in range(Nodes):
        for BC_type in BC_types:
            for quantity in Quantity_range:
                model = OneD(Elements, Nodes, k)
                model.K_stiffness()
                model.apply_boundary_condition(BC_type, node, quantity)
                solution = model.solve()
                if solution is not None:
                    # Flatten and zero-pad result if needed
                    temp_vec = solution.flatten()
                    padded_vec = np.pad(temp_vec, (0, Nodes - len(temp_vec)), 'cons
tant')
                    dataset.append(np.append([k, BC_type, node, quantity], padded_
vec))

# Convert to numpy array
```

```
dataset = np.array(dataset)

# Assemble final features with stiffness matrix
final_features = []
for row in dataset:
    k_val, bc_type, node, quantity = row[:4]
    temps = row[4:]

    model = OneD(Elements, Nodes, k_val)
    K_matrix = model.K_stiffness()
    flat_K = K_matrix.flatten()
    nonzero_flat_K = flat_K[flat_K != 0]

    feature_row = np.concatenate((temps, [k_val, bc_type, node, quantity], non
zero_flat_K))
    final_features.append(feature_row)

final_features = np.array(final_features)

# Save to CSV
np.savetxt("FEA_data.csv", final_features, delimiter=",")

# ==============================
# Machine Learning Model
# ==============================

# Prepare ML data
X = final_features[:, 10:]  # All features except T1 to T10
y = final_features[:, :10]  # T1 to T10 as regressors

# Define scalers for both X and y
scaler_X = StandardScaler()
scaler_y = StandardScaler()

# Scale the features (X)
X_scaled = scaler_X.fit_transform(X)
```

```
# Scale the target variables (y) for each output separately
y_scaled = scaler_y.fit_transform(y)

# Define pipeline with scaler and MLPRegressor
pipeline = Pipeline([
    ("scaler", StandardScaler()),  # This scales X during training
    ("mlp", MLPRegressor(max_iter=500))
])

# Define hyperparameter grid
param_grid = {
    "mlp__hidden_layer_sizes": [(50,)],  # Start simple
    "mlp__activation": ["relu", "tanh"],
    "mlp__solver": ["adam"],
    "mlp__alpha": [0.01, 0.1],
    "mlp__learning_rate": ["constant"],  # Can try "adaptive" later
    "mlp__learning_rate_init": [0.001],  # Small learning rate to start with
    "mlp__max_iter": [5000],  # Allow more iterations
    "mlp__early_stopping": [True],  # Stop early if not improving
    "mlp__batch_size": [32]  # Try with a lower batch size
}

# Set up GridSearchCV
grid_search = GridSearchCV(pipeline, param_grid, cv=3, scoring='neg_mean_
squared_error', verbose=2)

# Split the dataset: 80% training, 20% testing
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score

X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_scaled, test_size=0.
2, random_state=42)

# Fit model on training data
grid_search.fit(X_train, y_train)
```

```
# Evaluate on training data
y_train_pred = grid_search.predict(X_train)
train_mse = mean_squared_error(y_train, y_train_pred)
train_r2 = r2_score(y_train, y_train_pred)

# Evaluate on test data
y_test_pred = grid_search.predict(X_test)
test_mse = mean_squared_error(y_test, y_test_pred)
test_r2 = r2_score(y_test, y_test_pred)

# Inverse scale the predictions to get back to original units
y_train_pred_rescaled = scaler_y.inverse_transform(y_train_pred)
y_test_pred_rescaled = scaler_y.inverse_transform(y_test_pred)

# Best model and metrics
print("Best parameters:", grid_search.best_params_)
print("Best cross-validation score (neg MSE):", -grid_search.best_score_)
print(f"Training MSE: {train_mse:.4f}, R2: {train_r2:.4f}")
print(f"Test MSE: {test_mse:.4f}, R2: {test_r2:.4f}")
```

Gradient Boosting Model

```
import numpy as np
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.multioutput import MultiOutputRegressor
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.metrics import mean_squared_error, r2_score
import pandas as pd

# ============================
# FEA Batch Model
# ============================
class OneD:
    def __init__(self, Elements, Nodes, k):
        self.Elements = Elements
        self.Nodes = Nodes
        self.K_val = k
        self.K_matrix = np.zeros((self.Nodes, self.Nodes))
        self.Local_k = np.array([[1, -1], [-1, 1]])
        self.Force = np.zeros((self.Nodes, 1))

    def K_stiffness(self):
        for j in range(0, self.Nodes - 1):
            self.K_matrix[j:j+2, j:j+2] += self.K_val * self.Local_k
        return self.K_matrix

    def apply_boundary_condition(self, BC, Location, Quantity):
        if BC == 0:  # Essential (Dirichlet)
            self.Force[Location] += self.K_matrix[Location, Location] * Quantity
            self.K_matrix = np.delete(self.K_matrix, Location, axis=0)
            self.K_matrix = np.delete(self.K_matrix, Location, axis=1)
            self.Force = np.delete(self.Force, Location, axis=0)

        elif BC == 1:  # Natural (Neumann)
            self.Force[Location] += Quantity

        elif BC == 2:  # Mixed
            self.K_matrix[Location, Location] += self.K_val
            self.Force[Location] += self.K_val * Quantity

    def solve(self):
        try:
            T = np.linalg.solve(self.K_matrix, self.Force)
            return T
        except np.linalg.LinAlgError:
            return None

# Exhaustive generation parameters
```

```
Nodes = 10
Elements = Nodes - 1
k_values = np.linspace(5, 200, 100)  # Vary stiffness from 5 to 200
BC_types = [0, 1, 2]
Quantity_range = np.linspace(1, 150, 200)  #  values from 1 to 150

# Store results
dataset = []

# Loop through k values, all nodes, BC types and quantities
for k in k_values:
    for node in range(Nodes):
        for BC_type in BC_types:
            for quantity in Quantity_range:
                model = OneD(Elements, Nodes, k)
                model.K_stiffness()
                model.apply_boundary_condition(BC_type, node, quantity)
                solution = model.solve()
                if solution is not None:
                    # Flatten and zero-pad result if needed
                    temp_vec = solution.flatten()
                    padded_vec = np.pad(temp_vec, (0, Nodes - len(temp_vec)), 'cons
tant')
                    dataset.append(np.append([k, BC_type, node, quantity], padded_
vec))

# Convert to numpy array
dataset = np.array(dataset)

# Assemble final features with stiffness matrix
final_features = []
for row in dataset:
    k_val, bc_type, node, quantity = row[:4]
    temps = row[4:]

    model = OneD(Elements, Nodes, k_val)
    K_matrix = model.K_stiffness()
    flat_K = K_matrix.flatten()
    nonzero_flat_K = flat_K[flat_K != 0]

    feature_row = np.concatenate((temps, [k_val, bc_type, node, quantity], non
zero_flat_K))
    final_features.append(feature_row)

final_features = np.array(final_features)

# ==============================
# Machine Learning Model
# ==============================

# Prepare ML data
X = final_features[:, 10:]  # All features except T1 to T10
y = final_features[:, :10]  # T1 to T10 as regressors

# Scale target values (T1 to T10)
y_scaler = MinMaxScaler()
y = y_scaler.fit_transform(y)

# Split the dataset: 80% training, 20% testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_stat
e=42)

# Define pipeline with scaler and MultiOutputRegressor using GradientBoostin
g
pipeline = Pipeline([
    ("scaler", StandardScaler()),
    ("gb", MultiOutputRegressor(GradientBoostingRegressor()))
])

# Define hyperparameter grid
param_grid = {
    "gb__estimator__n_estimators": [100],
    "gb__estimator__learning_rate": [0.01],
    "gb__estimator__max_depth": [5],
    "gb__estimator__min_samples_split": [2],
    "gb__estimator__min_samples_leaf": [1],
    "gb__estimator__subsample": [0.8]
}

# Set up GridSearchCV
grid_search = GridSearchCV(pipeline, param_grid, cv=3, scoring='neg_mean_
squared_error', verbose=2)

# Fit model on training data
grid_search.fit(X_train, y_train)

# After predicting, inverse the scaling on the predictions
y_train_pred = grid_search.predict(X_train)
y_test_pred = grid_search.predict(X_test)

y_train_pred_inv = y_scaler.inverse_transform(y_train_pred)
y_test_pred_inv = y_scaler.inverse_transform(y_test_pred)

# Evaluate on training data
train_mse = mean_squared_error(y_train, y_train_pred)
train_r2 = r2_score(y_train, y_train_pred)

# Evaluate on test data
test_mse = mean_squared_error(y_test, y_test_pred)
test_r2 = r2_score(y_test, y_test_pred)

# Best model and metrics
print("Best parameters:", grid_search.best_params_)
print("Best cross-validation score (neg MSE):", -grid_search.best_score_)
print(f"Training MSE: {train_mse:.4f}, R2: {train_r2:.4f}")
print(f"Test MSE: {test_mse:.4f}, R2: {test_r2:.4f}")
```