Python Scientific Developer Assessment

Overview

The aim of this assessment is to evaluate the candidate's ability to efficiently comprehend and apply new scientific knowledge. Begin by studying the mathematical theory behind the finite element method, provided in the `beamer_FEM_presentation.pdf` file within the test folder. The primary task is to understand this material and subsequently code a basic implementation of the method.

Task

Examine the homogenous Poisson equation in a rectangular domain:

$$-\Delta u(x) = f(x) \qquad \forall\, x \in \Omega := [a,b] \times [c,d]$$

$$u(x) = 0 \qquad \forall\, x \in \partial\Omega$$

The objectives are to:

1. Implement the finite element method to resolve this equation using only essential libraries like scipy and numpy;
2. Analyze the method's convergence using an appropriate test case;
3. Visually display results with libraries such as matplotlib or plotly;
4. Develop an algorithm that refines any given mesh (i.e., adds nodes and elements), based on the convergence theory, to enhance solution accuracy.

Assessment Criteria

Candidates will be evaluated on:

- **Correctness** of the proposed solution;
- **Ability to vectorize** the code with numpy;
- Appropriateness in selecting data structures and methods for FEM;
- **Code structure**: Adherence to standard Python industry practices is highly encouraged. Simple "academic" Python scripting will not yield optimal scores, even if functional;

- **Result presentation**: Utilizing a notebook for proper presentation and commentary on results is recommended (main implementation should be in standard `.py` files).

Each step of the assessment aims to test your scientific coding ability and your understanding of the Python language. Good luck!

Some Clarifications Before Starting

The Finite Element Method (FEM) operates by solving a weak formulation in a finite subspace, formed by piecewise linear functions on elements. Each function in the basis is uniquely defined by a node and only possesses non-zero value within the surrounding elements of that node.

For practical implementation, it may be perplexing to learn that functions of the basis are not used directly due to their cumbersome nature. It is generally more effective to work directly with the elements. For instance, considering $a(v_i, v_j)$, its integral can be decomposed into the sum of the integrals over the elements shared by the nodes i and j:

$$a(v_i, v_j) = \int_\Omega \nabla v_i(x) \nabla v_j(x) dx = \sum_{\substack{E \in \mathcal{T} \\ \text{s.t.} \{x_i, x_j\} \subseteq E}} \int_E \nabla v_i(x) \nabla v_j(x) dx$$

Here, x_j and x_i are the nodes determining the functions v_j and v_i and E are the elements of the meshed domain Ω.

Every element contains three nodes, thus having three associated base functions. This denotes that each element contributes 9 entries to the stiffness matrix. We compute these contributions and incorporate them into a 3×3 matrix known as the local stiffness matrix. After adjusting the matrix indices, the global stiffness matrix A can be generated by aggregating contributions from each local stiffness matrix.

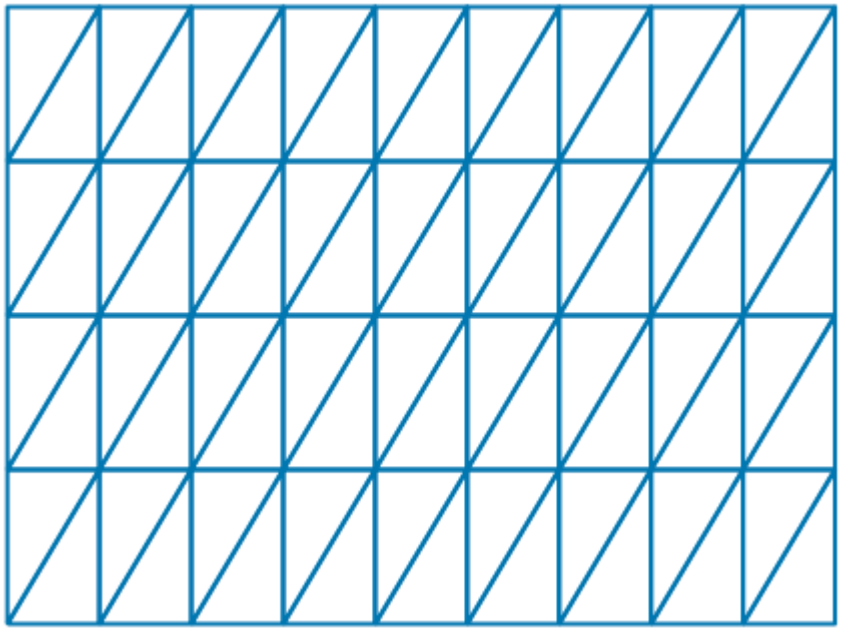
You can compute the local stiffness matrix on a simple element, and then perform a change of variable to generalize the computation to every possible triangle.

To enforce the zero boundary condition on node k, the recommended approach is setting $A_{k,k} = M$, where M is a significantly larger value and $b_k = 0$. A careful review of the theory might reveal an alternative way to enforce this condition - extra points for discovering it.

FEM Implementation Guidelines

For a successful FEM implementation, consider the following suggestions:

1. Establish a suitable mesh for the rectangular domain. Your setup should resemble this:



The code should allow an arbitrary number of subdivisions on the x and y axes (e.g., the image above shows four subdivisions along the y-axis). **Hint**: Meshes are typically represented using two matrices: a $N \times 2$ matrix for node positions and a $N \times 3$ matrix for the elements. Each row (say, i-th row) in the elements matrix defines the i-th element by referencing three nodes from the nodes matrix.

2. Construct the stiffness matrix $A = (a(v_j, v_i))_{i,j}$ for any given mesh. The presentation provides most of the implementation details. Notable considerations:
   - The global stiffness matrix A can be assembled by adding the contributions of each element's 3×3 local stiffness matrix;
   - Compute the local stiffness matrix for a basic simple element K where calculations are simple;
   - Apply a change of variables to determine the local stiffness matrix by transforming K into an alternate element $\tilde{K}$.
   - **Hint:** you can use this snippet instead if you don't know how to derive the formula. Obviously the complete deriviation is appreciated and contributes to a good evaluation:

```
def local_stiffness(element_nodes):
     area_of_the_triangle = ...
     # Variable name is bad on purpose
     sss = element_nodes[[2,0,1],:]-element_nodes[[1,2,0],:]
     return sss@sss.T / (4*area_of_the_triangle)
```

   - Modify the stiffness matrix to account for the boundary condition
3. Similarly, construct the right-hand side of the equation $b = (F(v_i))_i$. **Recommendation**: It's generally impossible to compute the integral represented by F exactly. You can use a simple

approximation formula to calculate this integral and maintain method convergence. If possible, explain the theoretical reasons behind this.

4. Lastly, solve the linear system related to the FEM.