Nicolino Primavera

FEM for Fluid Flow and FSI Interactions

Assignment 1

10/11/24

**Problem 3**

Objective:

* Using the Finite Element Method (FEM) to analyze the Steady-State Advection-Diffusion Equation
* Steady-state Advection-Diffusion Equation: A black text on a white background

  Description automatically generated with boundary conditions u(0) = 0 & u(1) = 10
* Want to explore the effect of cell Peclet number on the computed solution when Galerkin FEM is applied to this problem
* Pe number represents the ratio of advection to diffusion
* Use numerical integration or Gauss quadrature-based integration (Gauss integration)
* Solve:
  + a. Choose at least four different grids for a range of cell Peclet numbers (Pe) and demonstrate that oscillatory solutions are obtained when Pe > 2
  + b. Compare the solutions for Pe < 2 against the analytical solution

Methodology:

* Utilized Gauss Quadrature points and weights for numerical integration
* Solved a 2-point Gauss Quadrature that was used to integrate the local stiffness matrix contributions within each element by evaluating the shape functions and their derivatives at the quadrature points
* Computed the local stiffness matrix for a finite element of size h and Peclet number Pe in a 1D advection-diffusion problem
  + Computed the diffusion and advection term for each matrix entry
  + Compiled the local stiffness matrix
* Calculated the global stiffness matrix
  + The Global Stiffness Matrix represents the entire system of equations for all nodes in the domain [0,1[ - comprehensive assembly of contributions from each element's local stiffness matrix
  + Assembly: The local stiffness matrix for each element is computed and assembled into the global stiffness matrix by adding values to the corresponding rows and columns (nodes) of the global matrix
  + The Global Stiffness Matrix combines all the local element stiffness matrices, which creates a global system of equations that can be solved to approximate the solution of the advection-diffusion equation over the entire domain
  + Compute the Global Stiffness Matrix for the domain [0,1] with N nodes
  + Calculated by looping over the elements in the domain
* Calculated the grid size (h), number of nodes (N\_no), and nodal positions (x) based on the selected cell peclet numbers
* Assembled the global stiffness matrices from the local element matrices and initialized a solution vector ‘u’ to store the unknowns at each node
* Applied the boundary conditions and modified the global stiffness matrix to ensure the proper boundary conditions were enforced.
* Solved the system and plotted against the analytical solutions to prove that for Pe < 2 there are stable solutions and for Pe > 2 there are unstable solutions

Results and Discussion:

A graph of different colored lines

Description automatically generated

The figure above displays my code’s results. Peclet numbers below 2 have a smooth solution while peclet numbers above two have an oscillatory solution. This happens because convection and diffusion are not properly balanced in the numerical scheme (Galerkin Method). The oscillations in the solutions are due to numerical stability issues because convection and diffusion are not properly balanced in the numerical scheme. For convection dominated flows, the Galerkin method is unstable, but it is consistent and satisfies Galerkin orthogonality. The cell peclet number affects the accuracy of the Galerkin method. Large numbers (greater than two) are unstable and cause numerical oscillations.

Conclusion:

To conclude my findings, I can confidently say that oscillatory solutions are obtained when the peclet number is greater than two based on the plot my code generated. Once the peclet number was greater than two the solution transitioned from a smooth parabolic shape to an oscillatory triangular solution.

References:

Hughes, T. J. R. (2000). *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Dover Publications.

**Code:**

#!/usr/bin/env python3

"""

Nicolino Primavera

FEM for Fluid Flow and FSI Interactions

Assignment 1

10/11/24

Using the Finite Element Method (FEM) to analyze the Steady-State Advection-Diffusion Equation

Steady-state Advection-Diffusion Equation: (du/dx) = (0.05)\*(d^2u/dx^2) - strong form, Domain: 0 <= x <= 1 , u(0) = 0 & u(1) = 10

Want to explore the effect of cell Peclet number on the computed solution when Galerkin FEM is applied to this problem

Pe number represents the ratio of advection to diffusion

Use numerical inegration or Gauss quadrature-based integration (Gauss integration)

Solve:

a. Choose at least four different grids for a range of cell Peclet numbers (Pe) and demonstrate that oscillatory solutions are obtained when Pe > 2

b. Compare the solutions for Pe < 2 against the analytical solution

"""

print("\nStarting Program...\n")

import math

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

# Parameters

a = 1.0 # Advection speed

k = 0.05 # Diffusion coefficient

#Pe = (a\*h)/k = 20\*h # Cell peclet number

Pe\_values = [0.1, 0.5, 1, 1.5, 2, 2.5, 3, 4, 5, 6, 7] # Peclet numbers

#h = (Pe\*k)/a # Grid/cell spacing - use uniform spacing

# Domain

x\_0 = 0

x\_1 = 1

# Boundary Conditions

u\_0 = 0

u\_1 = 10

# Gauss Quadrature / Numerical Integration

def gauss\_quadrature(n\_points):

"""

Returns Gauss quadrature points and weights for numerical integration

Gauss Quadrature Formula: Integral from -1 to +1 of (f(ξ)\*dξ) = Summation from g=1 to # of int points (n) of f(ξ\_g)\*w\_g

f(ξ) - function being integrated

ξ\_g - quadrature points

w\_g - weights associated with each quadrature point , determine how much each function evaluation contributes to total integral

n - number of quadrature points

2 point Gauss Quadrature:

Quadrature points: ξ\_1 = -1/sqrt(3) , ξ\_2 = 1/sqrt(3) - these points are extremely accurate for linear and quadratic basis functions

Weights: w\_1 = w\_2 = 1

Use to integrate polynomials up to 3rd degree

Use quadrature points to integrate local stiffness matrix contributions within each element by evaluating the shape functions & their derivatives at the quadrature points

Increase the number of quadrature points to improve accuracy of if integrating a more complex function

"""

# Gauss Quadrature Rule for Numerical Integration

if n\_points == 2: # 2-point Gauss Quadrature

return np.array([-1/np.sqrt(3), 1/np.sqrt(3)]), np.array([1, 1]) # Returns 2-point quadrature points(2), weights

else:

raise ValueError("Only 2-point quadrature implemented.") # Only allows for 2-point Gauss Quadrature

# Stiffness Matrix

def local\_stiffness\_matrix(h, Pe):

"""

Compute the local stiffness matrix for a finite element of size h and Peclet number Pe in a 1D advection-diffusion problem

Compute the Diffusion Term for each matrix entry:

1/h - derivatives of the shape functions wrt physical coordinates

diffusion term is proportional to the inverse of the element size

when i == j the value is 1 (diagonal)

when 1 /= j the value is -1 (off-diagonal)

Compute the Advection Term for each matrix entry:

Pe - peclet number scales the contribution of advection relative to diffusion

w[i] - weight associated w/ Gauss quadrature point i

(0.5 \* (1 - xi[i]) - shape function N\_1(ξ) evaluated at quadrature point i

(0.5 \* (1 + xi[i]) - shape function N\_2(ξ) evaluated at quadrature point i

"""

K\_local = np.zeros((2, 2)) # Initialize Local Stiffness Matrix [2 x 2]

print(f"\nInitial Local Stiffness Matrix (should be 0): \n{K\_local}\n") # Error handling

xi, w = gauss\_quadrature(2) # Gauss quadrature points (xi) and weights (w)

print(f"Gauss Quadrature: {xi, w}") # Error handling

print(f"Quadrature points: {xi}") # Error handling

print(f"Weights: {w}\n") # Error handling

# Nested loop to fill the Local Stiffness Matrix [2 x 2]

for i in range(2):

for j in range(2):

# Diffusion term (du/dx) (constant over the element for linear basis functions)

diffusion = (1/h) \* (1 if i == j else -1) # Computes the Diffusion Term

print(f"Diffusion Term: {diffusion}") # Error handling

# Advection term (Pe \* u \* v)

advection = Pe \* w[i] \* (0.5 \* (1 - xi[i]) if i == 0 else 0.5 \* (1 + xi[i])) # Computes the Advection Term

print(f"Advection Term: {advection}") # Error handling

# Local Stiffness Matrix

K\_local[i, j] += diffusion + advection # Computes the Local Stiffness Matrix

print (f"Local Stiffness Matrix: \n{K\_local}\n") # Error handling

print (f"Local Stiffness Matrix: {K\_local}\n") # Error handling

return K\_local

# Global Stiffness Matrix

def assemble\_global\_matrices(N, h, Pe):

"""

The Global Stiffness Matrix represents the entire system of equations for all nodes in the domain [0,1[ - comprehensive assembly of contributions from each element's local stiffness matrix

Assembly: The local stiffness matrix for each element is computed and assembled into the global stiffness matrix by adding values to the corresponding rows and columns (nodes) of the global matrix

The Global Stiffness Matrix combines all the local element stiffness matrices, which creates a global system of equations that can be solved to approx the soln of the advection-diffusion equation over the entire domain

Compute the Global Stiffness Matrix for the domain [0,1] with N nodes

N - number of nodes in the finite element mesh

h - size of each element (distance b/w adjacent nodes), uniform for a structured grid

Pe - peclet number

"""

K\_global = np.zeros((N, N)) # Initialize an empty array of size [N x N] (nodes)

print(f"\nGlobal stiffness matrix (empty): \n{K\_global}\n") # Error handling - empty array of size [N x N]

# Loops over elements in the domain

for e in range(N-1): # Elements are iterated N-1 b/c there is one element b/w every two nodes, element number (e) goes from 0 to N-2

# Local stiffness matrix

K\_local = local\_stiffness\_matrix(h, Pe) # Assembling the local stiffness matrix [2 x 2] for each element - represents the interactions b/w the two nodes that define the element

print(f"Local Stiffness Matrix for {e}: \n{K\_local}\n") # Error handling

# Global stiffness matrix - assembling the local stiffness matrices into the corresponding part of the global stiffness matrix

K\_global[e:e+2, e:e+2] += K\_local # Indexing: for element e, the nodes involved are node e and node e+1 --> K\_global[e:e+2, e:e+2] refers to the 2 x 2 submatrix of K\_global that corresponds to these two nodes

print(f"Global Stiffness Matrix: \n{K\_global}\n") # Error handling

print(f"Global Stiffness Matrix: \n{K\_global}\n") # Error handling

return K\_global # Return the Global stiffness matrix

# Cell Peclet Number

for Pe in Pe\_values: # Iterate over selected Peclet numbers from above array

# Compute grid size based on Pe and define nodes

h = (Pe \* k) / a # Calculates grid size based on Pe

N\_no = int((x\_1 - x\_0) / h) + 1 # Computes number of nodes based on total domain length (x\_1 - x\_0)=1

x = np.linspace(x\_0, x\_1, N\_no) # Generates nodes positions , N\_no evenly spaced points b/w the domain boundaries --> positions of the nodes in the 1D domain

print(f"Grid size: {h}\n") # Error handling

print(f"Number of nodes: {N\_no}\n") # Error handling

print(f"Node positions:\n {x}\n") # Error handling

# Initialize global matrices

K\_global = assemble\_global\_matrices(N\_no, h, Pe) # Assembles the Global Stiffness Matrix from the local element matrices

u = np.zeros(N\_no) # Initialize solution vector 'u' - stores the unknowns at each node

print(f"Global Stiffness Matrix: \n{K\_global}\n") # Error handling

print(f"Solution vector 'u': \n{u}\n")

# Apply boundary conditions

u[0] = u\_0 # Sets first value of u at x=0 to the BC 0

u[-1] = u\_1 # Sets final value of u at x=1 to th BC 10

# Modify Global Stiffness Matrix to enforce BCs

K\_global[0, :] = 0 # Clears the first row

K\_global[0, 0] = 1 # Sets the first diagonal element to 1

K\_global[-1, :] = 0 # Clears the last row

K\_global[-1, -1] = 1 # Sets the last diagonal element to 1

# Solve the system

u = np.linalg.solve(K\_global, u) # Solves the system of linear eqns K\_global(u)=g

print(f"Solution vector 'u': \n{u}\n") # Error handling

# Plot

plt.plot(x, u, label=f'Pe = {Pe}')

# Alalytical Solution for comparison - satisfies BCs u(0)=0 and u(1)=10 --> u(x)=10x

x\_analytical = np.linspace(x\_0, x\_1, 100) # X values of analytical soln

u\_analytical = 10 \* x\_analytical # Linear solution from u(0) = 0 to u(1) = 10 (u(x) values)

plt.plot(x\_analytical, u\_analytical, 'k--', label='Analytical Solution') # Plot the analytical solution

# Plot

plt.xlabel('x')

plt.ylabel('u(x)')

plt.title('FEM Solutions for the Steady-State Advection-Diffusion Equation')

plt.legend()

plt.show()