Python and C++ extension

Importing library

In [1]:
```
import sys
sys.path.append('../../CppToPython')
```

In [2]:
```
import numpy as np
import GeDiM4Py as gedim
```

Initialize

In [3]:
```
lib = gedim.ImportLibrary("../../CppToPython/release/GeDiM4Py.so")

config = { 'GeometricTolerance': 1.0e-8 }
gedim.Initialize(config, lib)
```

Non-Linear Equation

Recap of what we did in theory during the Lesson n. 13.

Solving the following equation on square $\bar{\Omega} = [0,1] \times [0,1]$

$$\begin{cases} -\nabla \cdot (\nabla u) + u \nabla \cdot u = g & \text{in } \Omega \\ u = 0.0 & \text{in } \partial\Omega \text{ homogeneous Dirichlet boundary condition} \end{cases}$$

where $u = 16xy(1-x)(1-y)$.

The weak form of the problem becomes, find $u \in V := H^1_0(\Omega)$

$$\int_\Omega \nabla u \nabla v + \int_\Omega u \nabla \cdot u v - \int_\Omega g v = 0 \quad \forall v \in V \Leftrightarrow f(u;v) := f_1(u;v) + f_2(u;v) + f_3(u;v) = 0 \quad \forall v \in V$$

Using **Newton schema**, we solve for each k iteration the problem

$$J_f[\partial u]_{|_{u_k}} = -f(u_k; v) = 0 \quad \forall v \in V$$

where $J_f[\partial u]_{|_{u_k}}$ is the evaluation of the derivative (Jacobian) of J_f in the point u_k along the unknown direction of ∂u.

After computations, we find the linear system, on each k iteration, fixed u_k find ∂u s.t.

$$\int_\Omega \nabla \partial u \cdot \nabla v + \int_\Omega \nabla \cdot u_k \partial u \; v + \int_\Omega u_k \nabla \cdot \partial u \; v = -\int_\Omega \nabla u_k \cdot \nabla v - \int_\Omega \nabla u_k \cdot u_k \; v + \int_\Omega g v$$

Algorithm: Newton's Method for Nonlinear PDEs

1. **Input:**
   - Nonlinear PDE operator R
   - Tolerance ϵ
   - Maximum iterations m_{max}
   - Initial guess u_0.
2. **Output:** Approximate solution u.
3. Initialize $m = 0$.
4. Choose an initial guess u_0.
5. **For** $m = 0, 1, 2, \ldots, m_{max} - 1$:
   - Evaluate the residual $R(u_m)$. *Often in variational form: $F(u_m)(v)*
   - Evaluate the linearized operator $R'(u_m)$. *Often in variational form: find $J_F[\cdot](\cdot)$*
   - Solve the linear equation $R'(u_m)[\delta u_m] = -R(u_m)$ for the correction δu_m. *Often in variational form: $J_F[\delta u_m](v) = -F(u_m)(v)$*
   - Update the solution: $u_{m+1} = u_m + \delta u_m$.
   - Check for convergence.
   - **If** $||R(u_{m+1})|| < \epsilon$ or $||\delta u_m|| < \epsilon$:
     - Set $u = u_{m+1}$.
     - **break** (Exit the loop)
6. **If** $m == m_{max} - 1$ and convergence not reached: **Warning:** Maximum number of iterations reached without convergence.
7. Set $u = u_{m+1}$.
8. **Return:** u.

Recap

Function	Represents	Mathematical Expression
`Burger_f`	Source term `g`	`g(x, y)`
`Burger_non_linear_f`	Nonlinear term	`u * div(u)`
`Burger_non_linear_der_f`	Derivative (w.r.t. u) of nonlinear term	`grad(u)` ≈ `[∂u/∂x, ∂u/∂y]`

In [ ]:
```python
# points := list of quadrature points x_1, x_2, ..., x_q

# Do not depends on u_N, hence on the previous iteration
def Burger_a(numPoints, points): # Diffusion (1 integral in the math left part)
        values_a = np.ones(numPoints, order='F')
        return values_a.ctypes.data

def Burger_b(numPoints, points): # Advection
        values_b = np.ones((2, numPoints), order='F')
        return values_b.ctypes.data

def Burger_c(numPoints, points): # Reaction
        values_c = np.ones(numPoints, order='F')
        return values_c.ctypes.data

# Here in the code, every time that we see name_variable_non_linear, it means that the variable is related to the
# non-linear part of the equation
# Then, this variable depends on u_N
def Burger_non_linear_b(numPoints, points, u, u_x, u_y): # Advection parameter related to the previous iteration
                                                                                # (3 integral in the math left part)
        vecu = gedim.make_nd_array(u, numPoints, np.double) # Evaluation of the function in some points
        values_nl_b = vecu
        return values_nl_b.ctypes.data

def Burger_non_linear_c(numPoints, points, u, u_x, u_y): # Reaction integral in which the parameter depends on u_N
                                                                                # (2 integral ini the math left part)
        vecu_x = gedim.make_nd_array(u_x, numPoints, np.double) # Derivative of x := [d_x u_N|x_1, ..., d_x u_N|x_Q]
                    # Hence, it's an array that contains the derivarive wrt x of u_N computed in the different x_i
                                        # quadrature point
        vecu_y = gedim.make_nd_array(u_y, numPoints, np.double) # Derivative of y --> the same but wrt y
        values_nl_c = vecu_x + vecu_y # Summation of the derivative (this represents the "strange" divergence)
        return values_nl_c.ctypes.data

def Burger_f(numPoints, points): # Right hand side (3 integral in the math right part)
        matPoints = gedim.make_nd_matrix(points, (3, numPoints), np.double)
        values_f = 32.0 * (matPoints[1,:] * (1.0 - matPoints[1,:]) + matPoints[0,:] * (1.0 - matPoints[0,:])) + \
        (16.0 * (1.0 - 2.0 * matPoints[0,:]) * matPoints[1,:] * (1.0 - matPoints[1,:]) + \
        16.0 * (1.0 - 2.0 * matPoints[1,:]) * matPoints[0,:] * (1.0 - matPoints[0,:])) * \
        16.0 * (matPoints[1,:] * (1.0 - matPoints[1,:]) * matPoints[0,:] * (1.0 - matPoints[0,:]))
        return values_f.ctypes.data

def Burger_non_linear_f(numPoints, points, u, u_x, u_y):
        vecu = gedim.make_nd_array(u, numPoints, np.double)
        vecu_x = gedim.make_nd_array(u_x, numPoints, np.double)
        vecu_y = gedim.make_nd_array(u_y, numPoints, np.double)
        values_nl_f = vecu * (vecu_x + vecu_y)
        return values_nl_f.ctypes.data

def Burger_non_linear_der_f(numPoints, points, u, u_x, u_y):
        vecu_x = gedim.make_nd_array(u_x, numPoints, np.double)
        vecu_y = gedim.make_nd_array(u_y, numPoints, np.double)
        values_nl_d_f = np.zeros((2, numPoints), order='F')
        values_nl_d_f[0,:] = vecu_x
        values_nl_d_f[1,:] = vecu_y
        return values_nl_d_f.ctypes.data

def Burger_exactSolution(numPoints, points): # Exact solution u = 16 xy(1-x)(1-y)
        matPoints = gedim.make_nd_matrix(points, (3, numPoints), np.double)
        values_ex = 16.0 * (matPoints[1,:] * (1.0 - matPoints[1,:]) * matPoints[0,:] * (1.0 - matPoints[0,:]))
        return values_ex.ctypes.data

def Burger_exactDerivativeSolution(direction, numPoints, points):
        matPoints = gedim.make_nd_matrix(points, (3, numPoints), np.double)

        if direction == 0:
                values_ex_d = 16.0 * (1.0 - 2.0 * matPoints[0,:]) * matPoints[1,:] * (1.0 - matPoints[1,:])
        elif direction == 1:
                values_ex_d = 16.0 * (1.0 - 2.0 * matPoints[1,:]) * matPoints[0,:] * (1.0 - matPoints[0,:])
        else:
                values_ex_d = np.zeros(numPoints, order='F')

        return values_ex_d.ctypes.data

def Ones(numPoints, points):
        values_one = np.ones(numPoints, order='F')
        return values_one.ctypes.data

def OnesDerivative(numPoints, points):
        values_one_d = np.ones((2, numPoints), order='F')
        return values_one_d.ctypes.data

def Zeros(numPoints, points):
        values_zero = np.zeros(numPoints, order='F')
        return values_zero.ctypes.data
```

```python
def ZerosDerivative(direction, numPoints, points):
        values_zero_d = np.zeros(numPoints, order='F')
        return values_zero_d.ctypes.data
```

Define Simulation Parameters

Set geometry parameters

In [ ]:
```python
meshSize = 0.01
order = 1

# Discrete space - finite element order 1
domain = { 'SquareEdge': 1.0, 'VerticesBoundaryCondition': [1,1,1,1], 'EdgesBoundaryCondition': [1,1,1,1],
          'DiscretizationType': 1, 'MeshCellsMaximumArea': meshSize }
[meshInfo, mesh] = gedim.CreateDomainSquare(domain, lib)

discreteSpace = { 'Order': order, 'Type': 1, 'BoundaryConditionsType': [1, 2] }
[problemData, dofs, strongs] = gedim.Discretize(discreteSpace, lib)
```

Set Newton parameters

In [ ]:
```python
# Variable using in the newton iteration
residual_norm = 1.0
solution_norm = 1.0;
newton_tol = 1.0e-6 # Tolerance for the stopping criteria
max_iterations = 7 # We do not know if the Newton Scheme converges or not --> stop at a certain point
                # Remember that Newton goverges iif the starting point is close enough to the solution
num_iteration = 1
```

Set Initial Solution

In [ ]:
```python
# Initialization of the guess of the solution --> the NM converges rapidly if we start not far away from the solution
u_k = np.zeros(problemData['NumberDOFs'], order='F') # Starting from 0, not a good idea to the speed of convergence
                            # Consider that for the project it's better to start from a better guess of the solution

u_strong = np.zeros(problemData['NumberStrongs'], order='F')
```

Run Newton Algorithm

Using a **relative tolerance**, we have to compute relative error (in notes)

In [ ]:
```python
while num_iteration < max_iterations and residual_norm > newton_tol * solution_norm: # We select a relative tollerance!
                                                                                  # We have to compute relative error to evaluate that
    [stiffness, stiffnessStrong] = gedim.AssembleStiffnessMatrix(Burger_a, problemData, lib) # Linear

    # Non linear - we need the previuous iteration evaluation
    # Hence, here we have to add the parameters Burger_non_linear_c for the reaction
    #                                           Burger_non_linear_b for the advection
    [reaction, reactionStrong] = gedim.AssembleNonLinearReactionMatrix(Burger_c, Burger_non_linear_c, u_k, u_strong, problemData, lib)
    [advection, advectionStrong] = gedim.AssembleNonLinearAdvectionMatrix(Burger_b, Burger_non_linear_b, u_k, u_strong, problemData, lib)

    # Right hand side of the function
        # Linear part
    forcingTerm_g = gedim.AssembleForcingTerm(Burger_f, problemData, lib)
        # Non linear part
    forcingTerm_v = gedim.AssembleNonLinearForcingTerm(Ones, Burger_non_linear_f, u_k, u_strong, problemData, lib)
    forcingTerm_der_v = gedim.AssembleNonLinearDerivativeForcingTerm(OnesDerivative, Burger_non_linear_der_f, u_k, u_strong,
                                                                     problemData, lib)

    # Solving with the LU solver because we're in a generic setting (no idea of the structure of the matrix)
    du = gedim.LUSolver(stiffness + advection + reaction, \
            forcingTerm_g - forcingTerm_v - forcingTerm_der_v, \
            lib)

    u_k = u_k + du

    du_normL2 = gedim.ComputeErrorL2(Zeros, du, np.zeros(problemData['NumberStrongs'], order='F'), lib)

    # Compute the error if we have the exact solution --> because we know the exact solution
    u_errorL2 = gedim.ComputeErrorL2(Burger_exactSolution, u_k, u_strong, lib)
    u_errorH1 = gedim.ComputeErrorH1(Burger_exactDerivativeSolution, u_k, u_strong, lib)

    # Compute the norm if we do not have the exact solution
    u_normL2 = gedim.ComputeErrorL2(Zeros, u_k, u_strong, lib)
    u_normH1 = gedim.ComputeErrorH1(ZerosDerivative, u_k, u_strong, lib)

    solution_norm = u_normL2;
    residual_norm = du_normL2;

    print("dofs", "h", "errorL2", "errorH1", "residual", "iteration", "max_iteration")
    print(problemData['NumberDOFs'], '{:.16e}'.format(problemData['H']), '{:.16e}'.format(u_errorL2 / u_normL2),
          '{:.16e}'.format(u_errorH1 / u_normH1), '{:.16e}'.format(residual_norm / u_normL2),
            '{:d}'.format(num_iteration), '{:d}'.format(max_iterations))

    num_iteration = num_iteration + 1
```

```
The history saving thread hit an unexpected error (OperationalError('attempt to write a readonly database')).History will not be written to th
e database.
dofs h errorL2 errorH1 residual iteration max_iteration
57 2.0647876100132428e-01 5.2260777660062499e-02 1.9159453441746624e-01 1.0000000000000000e+00 1 7
dofs h errorL2 errorH1 residual iteration max_iteration
57 2.0647876100132428e-01 3.5051158056069606e-02 1.8309858647719299e-01 3.4100994920636923e-02 2 7
dofs h errorL2 errorH1 residual iteration max_iteration
57 2.0647876100132428e-01 3.5050027767402056e-02 1.8309836416946049e-01 2.4831242817296458e-05 3 7
dofs h errorL2 errorH1 residual iteration max_iteration
57 2.0647876100132428e-01 3.5050027765334148e-02 1.8309836416908298e-01 8.0326321390164110e-12 4 7
```

Plot Solution

In [ ]:
```python
# Plot and show the solution
gedim.PlotSolution(mesh, dofs, strongs, u_k, u_strong)
gedim.ExportSolution(Burger_exactSolution, u_k, u_strong, lib)

[numQuadraturePoints, quadraturePoints, quadratureWeights, sol, sol_x, sol_y] = gedim.EvaluateSolutionOnPoints(u_k, u_strong, lib)
gedim.ExportSolutionOnPoints(numQuadraturePoints, quadraturePoints, sol, lib) # To export solutions on paraview

# EvaluateSolutionOnPoints --> this function (taking from gedim) let you evaluate the solution, given the degree of fredom (u_k)
#  and u_strong, in all the point of the
# mesh (the tasselation), we need this quantity to evaluate the error, or using in the training phase of the NN

# IMPORTANT FOR THE PROJECT: if you have a PINN, and you want to evaluate your PINN vs a finite element solution,
# you can use these points to do the comparison

# The quadrature formula is related to the order or your finite element
# The quadrature points are the points where you evaluate the solution, and the weights are the weights of the quadrature formula

# The higer is the degree of the finite element space, the more points you have in the quadrature formula
```