Finite element simulation of the Poisson problem on a cube skin

M. Ndjinga, M. Nguemfouo

May 30, 2021

1 Introduction

We consider the unit cube $\Omega_{cube} = [0,1] \times [0,1] \times [0,1] \subset \mathbb{R}^3$ and its boundary

$$\Gamma = \partial\Omega_{cube}.$$

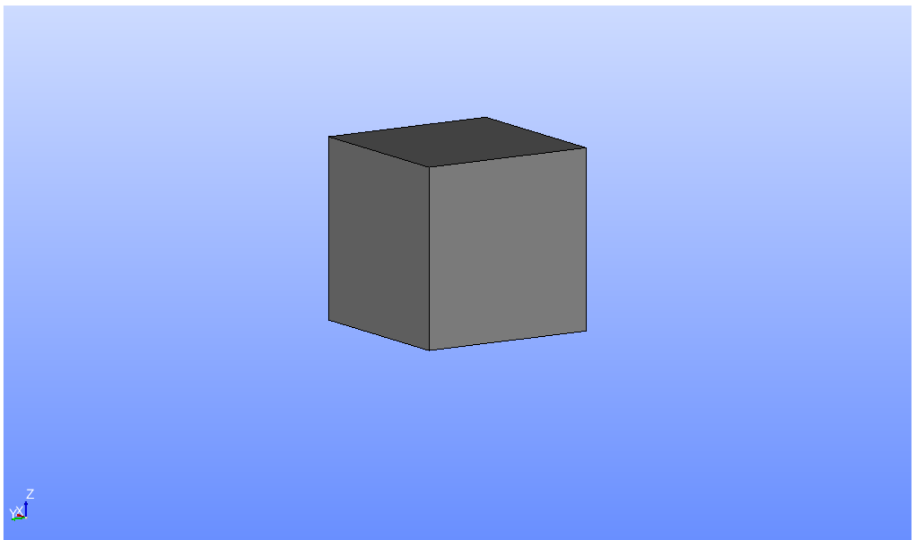


Figure 1: The unit cube in SALOME CAO module

Γ is a topological manifold but not a differential manifold because of the presence of sharp edges where Γ admits no tangent space. Γ is however a Lipschitz manifold, an intermediate structure between topological and differentiable manifolds [3, 4, 5], which is only differentiable almost everywhere thanks to Rademacher theorem. As a Lipschitz manifold, Γ can be endowed with a Laplace-Beltrami operator $\triangle_\Gamma = \nabla \cdot \vec{\nabla}$, defined as the combination of a surface divergence $\nabla_\Gamma \cdot$, and of a surface gradient $\vec{\nabla}_\Gamma$ (see [3]).

We consider f the restriction of the smooth function $\cos(2\pi x)\cos(2\pi y)\cos(2\pi z)$ on Γ. The explicit expressions of f on each face of Γ are

$$f(x,y,z) = \begin{cases} \cos(2\pi x)\cos(2\pi y) & \text{if} \quad z=0 \text{ or } z=1 \\ \cos(2\pi x)\cos(2\pi z) & \text{if} \quad y=0 \text{ or } y=1 \\ \cos(2\pi y)\cos(2\pi y) & \text{if} \quad x=0 \text{ or } x=1 \end{cases}.$$

f is an eigenfunction of the Laplace-beltrami operator on Γ since

$$\begin{aligned} \triangle_\Gamma f(x,y,z) &= \begin{cases} \partial_{xx}\cos(2\pi x)\cos(2\pi y) + \partial_{yy}\cos(2\pi x)\cos(2\pi y) & \text{if} \quad z=0 \text{ or } z=1 \\ \partial_{xx}\cos(2\pi x)\cos(2\pi z) + \partial_{zz}\cos(2\pi x)\cos(2\pi z) & \text{if} \quad y=0 \text{ or } y=1 \\ \partial_{yy}\cos(2\pi y)\cos(2\pi z) + \partial_{zz}\cos(2\pi y)\cos(2\pi z) & \text{if} \quad x=0 \text{ or } x=1 \end{cases} \\ &= \begin{cases} -2(2\pi)^2\cos(2\pi x)\cos(2\pi y) & \text{if} \quad z=0 \text{ or } z=1 \\ -2(2\pi)^2\cos(2\pi x)\cos(2\pi z) & \text{if} \quad y=0 \text{ or } y=1 \\ -2(2\pi)^2\cos(2\pi y)\cos(2\pi y) & \text{if} \quad x=0 \text{ or } x=1 \end{cases} \\ &= -8\pi^2 f. \end{aligned}$$

We are going to solve the following Poisson problem on Γ :

$$-\triangle_\Gamma u = f, \qquad (1)$$

$$\int_\Gamma u = 0,$$

where the right hand side $f \in L^2(\Gamma)$ and the unknown $u \in H^1(\Gamma)$ are **zero mean functions**.

Our objective is to solve numerically the Poissson problem (1) using the finite element method described in [10].

2 Finite elements method for 3D Poisson problem

2.1 Existence and uniqueness of the solution

Since Γ is closed (no boundary), we have to impose the global condition $\int_\Gamma u = 0$ to guarantee the uniqueness of solutions (otherwise, constants would be in the kernel of the Laplace-Beltrami operator).
Following [9] we define the following Lebesgue space

$$L^2_\#(\Gamma) = \{w \in L^2(\Gamma) : \int_\Gamma w = 0\}$$

and the following Sobolev space

$$H^1_\#(\Gamma) = \{w \in H^1(\Gamma) : \int_\Gamma w = 0\}.$$

These spaces are well defined on (non smooth) Lipschitz manifolds such as the unit cube, where the tangent space exists almost everywhere but not everywhere.

2.1.1 Variational formulation

Thanks to the **Green-Ostrograski** theorem for Lipschitz manifolds proved in [9], the variational formulation of (1) is:

$$\text{find } u \in H^1_{\#}(\Gamma) \text{ such that } \forall v \in H^1_{\#}(\Gamma), \int_\Gamma \overrightarrow{\nabla}_\Gamma u \cdot \overrightarrow{\nabla}_\Gamma v = \int_\Gamma fv \qquad (2)$$

2.1.2 Existence of the weak solution

The bilinear form

$$a(u,v) = \int_\Gamma \overrightarrow{\nabla}_\Gamma u \cdot \overrightarrow{\nabla}_\Gamma v$$

is continuous and coercive thanks to **Poincaré inequality** for Lipschitz manifolds proved in [9].
The linear form

$$b(v) = \int_\Gamma fv$$

is continuous.
By application of the **Lax-Milgram theorem**, the variational formulation (2) of problem (1) admits a unique weak solution. Furthermore, this solution depends continuously of the data f.
Due to the lack of smoothness of Γ, the regularity results from [7] (theorem theorem 3.3) is no longer valid, since it requires the regularity of both the right hand side and the manifold (C^3 manifold for the definition of Fermi coordinates and lift operator).

2.2 The P1 finite element for Poisson problem

Following [7], we first approximate the sphere Γ by a polyhedral surface Γ_h with triangular faces $(\mathcal{T}_k)_{k\geq 1}$ called elements having their nodes on Γ. We approximate functions $f \in H^1_{\#}(\Gamma)$ by functions $f_h \in H^1_{\#}(\Gamma_h)$. Due to the lack of regularity of the manifold, we cannot define precisely f_h using the lift operator as in [6, 7] since it requires a C^3 manifold. Also we will not be able to perform a convergence analysis similar to the one in [6, 7].

We consider the weak Laplace-Beltrami operator on the piecewise linear manifold Γ_h. Then we look for $\tilde{u}_h$ the projection of the solution $u_h \in H^1_{\#}(\Gamma_h)$ of the new Poisson problem

$$-\triangle_{\Gamma_h} \tilde{u}_h = f_h,$$

set on the space of continuous piecewise affine functions with zero mean $V_0(\Gamma_h)$. The discrete form of the variational formulation (2) is given by.

$$\text{Find } \tilde{u}_h \in V_0(\Gamma_h) \text{ such that } \forall \tilde{v}_h \in V_0(\Gamma_h), \int_{\Gamma_h} \overrightarrow{\nabla}_{\Gamma_h} \tilde{u}_h \cdot \overrightarrow{\nabla}_{\Gamma_h} \tilde{v}_h = \int_{\Gamma_h} f_h \tilde{v}_h. \quad (3)$$

Since $V_0(\Gamma_h)$ is generated by the nodal functions $\phi_i : \Gamma_h \to \mathbb{R}, \quad i = 1, ..., n$ such that $\phi_i(x_j) = \delta_{ij}$, (3) takes the following algebraic form

$$A_{\triangle_{\Gamma_h}} X = b_h, \qquad (4)$$

where

$$\tilde{u}_h = \sum_{i=1}^{n} u_i \phi_i, \tag{5}$$

$A_{\triangle_{\Gamma_h}} = (a_{ij})_{i,j=1,...,n}$, $X = {}^t(u_1, ..., u_n)$ and $b_h = {}^t(b_1, ..., b_n)$ with

$$a_{ij} = \int_{\Gamma_h} \overrightarrow{\nabla}_{\Gamma_h} \phi_i \cdot \overrightarrow{\nabla}_{\Gamma_h} \phi_j = \sum_{k=1}^{n} \int_{\mathcal{T}_k} \overrightarrow{\nabla}_{\Gamma_h} \phi_i \cdot \overrightarrow{\nabla}_{\Gamma_h} \phi_j,$$

$$b_j = \int_{\Gamma_h} f \phi_j = \sum_{k=1}^{n} \int_{\mathcal{T}_k} f \phi_j.$$

$A_{\triangle_{\Gamma_h}}$ is symmetric positive and sparse but not invertible since constants are in its kernel, hence the linear system (4) is singular. However it admits a unique solution with zero mean provided the right hand side has zero mean (see [7]).

3 Numerical results for Laplace-Beltrami operator on the unit cube skin

For the numerical resolution of our discrete problem, we use an iterative solver because the stiffness matrix $A_{\triangle_{\Gamma_h}}$ is large and sparse (see [8]) .

For the design and meshing of the domain we use GEOMETRY and MESH modules of the software SALOME 9.5 (see [11, 12]).

For the visualization of the result, we use the PARAVIS module included in SALOME (see [12]).

For the coding of the script, we use Python with the open-source Linux based library SOLVERLAB [13] which is very practical for the manipulation of large matrices, vectors, meshes and fields. It (SOLVERLAB) can handle finite element and finite volume discretizations, read general $1D, 2D$ and $3D$ geometries and meshes generated by SALOME.

3.1 Meshing of the domain

Below are the meshes used in our convergence analysis.

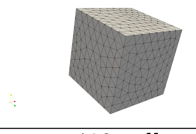
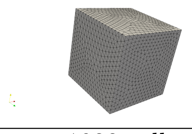
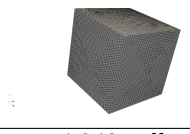
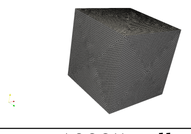
meshCubeSkin 1	meshCubeSkin 2	meshCubeSkin 3	meshCubeSkin 4
			
412 cells	1923 cells	7042 cells	13225 cells

Figure 2: Meshes of the unit cube skin

3.2 Visualization of the results

Below are visualizations of the numerical results obtained on the different meshes

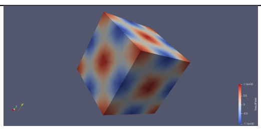
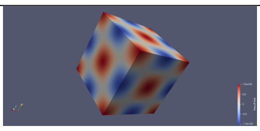
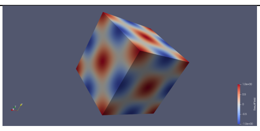
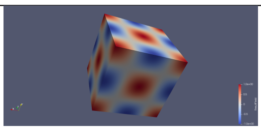
			
412 cells	1923 cells	7042 cells	13225 cells

Figure 3: Numerical results of the finite elements on the unit cube skin

Below are clipings of the previous numerical results.

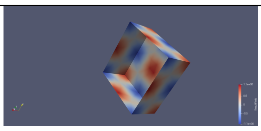
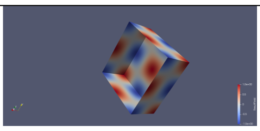
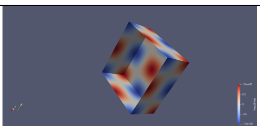
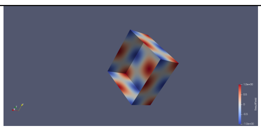
			
412 cells	1923 cells	7042 cells	13225 cells

Figure 4: Cliping of the numerical result on the unit cube skin

3.3 Numerical convergence of the finite element method

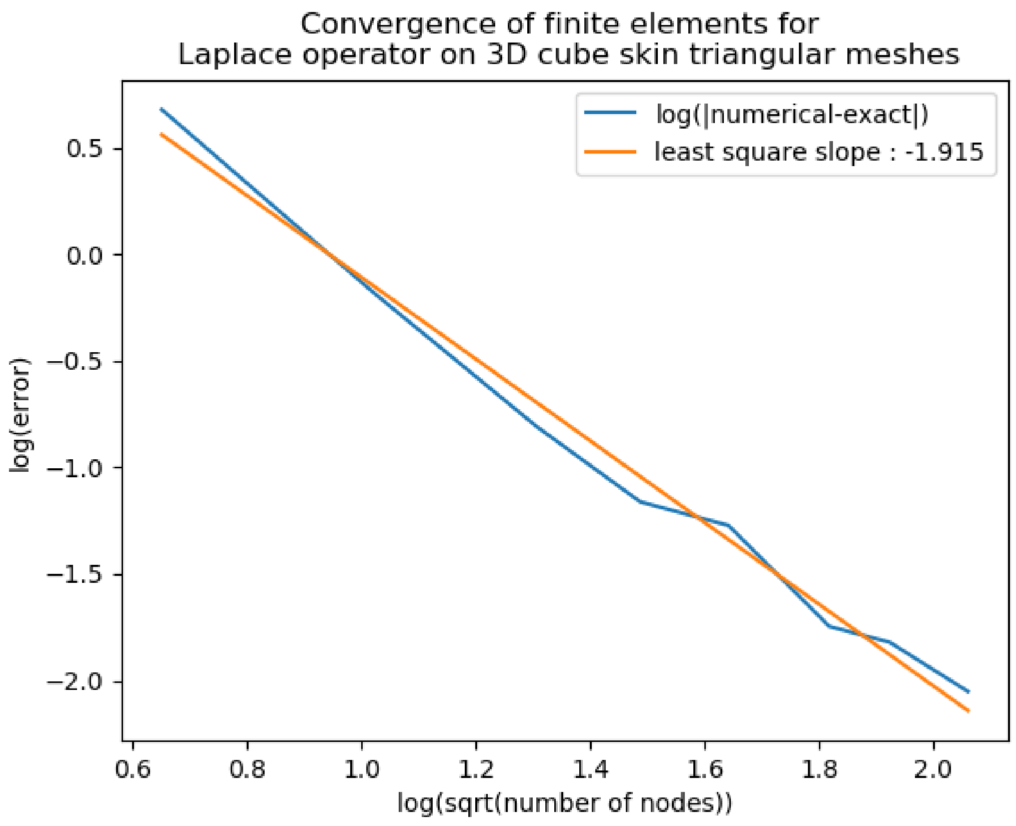


Figure 5: Convergence of the finite element method on the cube skin

The method converges with a numerical order of approximately 1.91.

3.4 Computational time of the finite element method

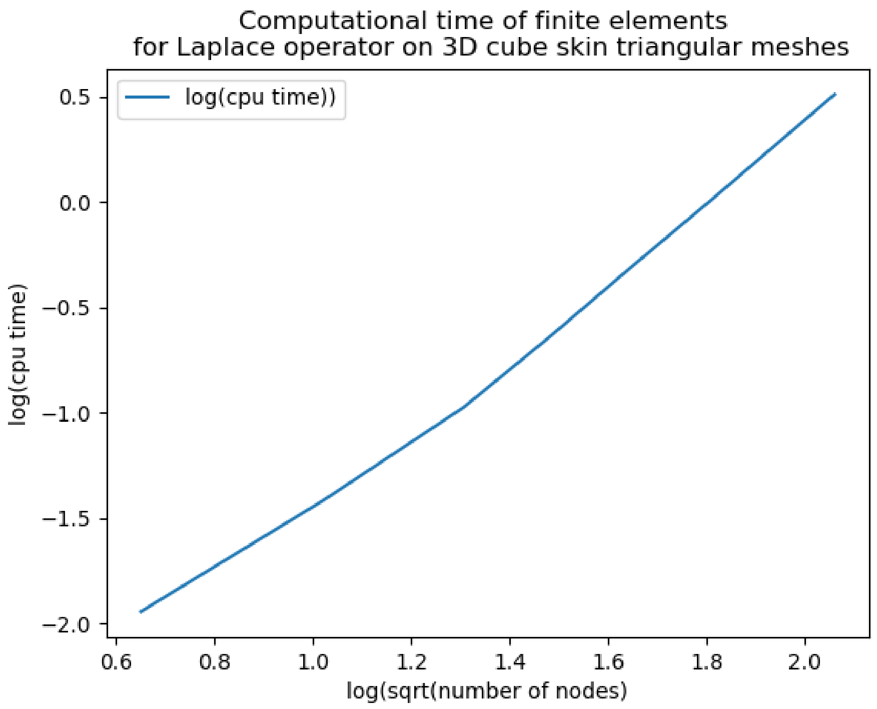


Figure 6: Computational time of the finite element method on the cube skin

4 The script

```
# -*-coding:utf-8 -*
#========================================================================================
# Name        : Résolution EF de l'équation de Laplace-Beltrami -\triangle u = f sur la frontière d'un cube
# Author      : Michael Ndjinga
# Copyright   : CEA Saclay 2021
# Description : Utilisation de la méthode des éléménts finis P1 avec champs u et f discrétisés aux noeuds d'un maillage triangulaire
#               Création et sauvegarde du champ résultant ainsi que du champ second membre en utilisant la librairie CDMATH
#               Résolution d'un système linéaire à matrice singulière : les vecteurs constants sont dans le noyau
#               Comparaison de la solution numérique avec la solution exacte définie face par face : u(x,y,z)= cos(2*pi*x)*cos(2*pi*y)*cos(2*pi*z)
#========================================================================================

```

```python
import cdmath
from math import cos, pi
import numpy as np
import PV_routines
import VTK_routines
import paraview.simple as pvs

#Chargement du maillage triangulaire de la frontière du cube unité [0,1]x[0,1]x[0,1]
#=======================================================================================================================

my_mesh = cdmath.Mesh("meshCubeSkin.med")
if(not my_mesh.isTriangular()) :
  raise ValueError("Wrong cell types : mesh is not made of triangles")
if(my_mesh.getMeshDimension()!=2) :
  raise ValueError("Wrong mesh dimension : expected a surface of dimension 2")
if(my_mesh.getSpaceDimension()!=3) :
  raise ValueError("Wrong space dimension : expected a space of dimension 3")

nbNodes = my_mesh.getNumberOfNodes()
nbCells = my_mesh.getNumberOfCells()

print("Mesh building/loading done")
print("nb of nodes=", nbNodes)
print("nb of cells=", nbCells)

#Discrétisation du second membre et détermination des noeuds intérieurs
#======================================================================

my_RHSfield = cdmath.Field("RHS field", cdmath.NODES, my_mesh, 1)
maxNbNeighbours = 0#This is to determine the number of non zero coefficients in the sparse finite element rigidity matrix

eps=1e-6
#parcours des noeuds pour discrétisation du second membre et extraction du nb max voisins d'un noeud
for i in range(nbNodes):
  Ni=my_mesh.getNode(i)
  x = Ni.x()
  y = Ni.y()
  z = Ni.z()

  my_RHSfield[i]= 8*pi*pi*cos(2*pi*x)*cos(2*pi*y)*cos(2*pi*z)

  if my_mesh.isBorderNode(i): # Détection des noeuds frontière
    raise ValueError("Mesh should not contain borders")
  else:
    maxNbNeighbours = max(1+Ni.getNumberOfCells(),maxNbNeighbours) #true only for planar cells, otherwise use function Ni.getNumberOfEdges()

print("Right hand side discretisation done")
print("Max nb of neighbours=", maxNbNeighbours)
print("Integral of the RHS", my_RHSfield.integral(0))

# Construction de la matrice de rigidité et du vecteur second
```

```
    membre du système linéaire
#
    ==================================================================================
Rigidite=cdmath.SparseMatrixPetsc(nbNodes,nbNodes,maxNbNeighbours)#
     warning : third argument is number of non zero coefficients
    per line
RHS=cdmath.Vector(nbNodes)

# Vecteurs gradient de la fonction de forme associée à chaque noeud
     d'un triangle
GradShapeFunc0=cdmath.Vector(3)
GradShapeFunc1=cdmath.Vector(3)
GradShapeFunc2=cdmath.Vector(3)

normalFace0=cdmath.Vector(3)
normalFace1=cdmath.Vector(3)

#On parcourt les triangles du domaine
for i in range(nbCells):

  Ci=my_mesh.getCell(i)

  #Contribution à la matrice de rigidité
  nodeId0=Ci.getNodeId(0)
  nodeId1=Ci.getNodeId(1)
  nodeId2=Ci.getNodeId(2)
  N0=my_mesh.getNode(nodeId0)
  N1=my_mesh.getNode(nodeId1)
  N2=my_mesh.getNode(nodeId2)

  #Build normal to cell Ci
  normalFace0[0]=Ci.getNormalVector(0,0)
  normalFace0[1]=Ci.getNormalVector(0,1)
  normalFace0[2]=Ci.getNormalVector(0,2)
  normalFace1[0]=Ci.getNormalVector(1,0)
  normalFace1[1]=Ci.getNormalVector(1,1)
  normalFace1[2]=Ci.getNormalVector(1,2)

  normalCell = normalFace0.crossProduct(normalFace1)
  normalCell = normalCell*(1/normalCell.norm())

  cellMat=cdmath.Matrix(4)
  cellMat[0,0]=N0.x()
  cellMat[0,1]=N0.y()
  cellMat[0,2]=N0.z()
  cellMat[1,0]=N1.x()
  cellMat[1,1]=N1.y()
  cellMat[1,2]=N1.z()
  cellMat[2,0]=N2.x()
  cellMat[2,1]=N2.y()
  cellMat[2,2]=N2.z()
  cellMat[3,0]=normalCell[0]
  cellMat[3,1]=normalCell[1]
  cellMat[3,2]=normalCell[2]
  cellMat[0,3]=1
  cellMat[1,3]=1
  cellMat[2,3]=1
  cellMat[3,3]=0

  #Formule des gradients voir EF P1 -> calcul déterminants
  GradShapeFunc0[0]= cellMat.partMatrix(0,0).determinant()*0.5
  GradShapeFunc0[1]=-cellMat.partMatrix(0,1).determinant()*0.5
  GradShapeFunc0[2]= cellMat.partMatrix(0,2).determinant()*0.5
  GradShapeFunc1[0]=-cellMat.partMatrix(1,0).determinant()*0.5
  GradShapeFunc1[1]= cellMat.partMatrix(1,1).determinant()*0.5
  GradShapeFunc1[2]=-cellMat.partMatrix(1,2).determinant()*0.5
  GradShapeFunc2[0]= cellMat.partMatrix(2,0).determinant()*0.5
  GradShapeFunc2[1]=-cellMat.partMatrix(2,1).determinant()*0.5
  GradShapeFunc2[2]= cellMat.partMatrix(2,2).determinant()*0.5

  #Création d'un tableau (numéro du noeud, gradient de la fonction
    de forme
  GradShapeFuncs={nodeId0 : GradShapeFunc0}
  GradShapeFuncs[nodeId1]=GradShapeFunc1
  GradShapeFuncs[nodeId2]=GradShapeFunc2

  # Remplissage de  la matrice de rigidité et du second membre
  for j in [nodeId0,nodeId1,nodeId2] :
    #Ajout de la contribution de la cellule triangulaire i au
    second membre du noeud j
    RHS[j]=Ci.getMeasure()/3*my_RHSfield[j]+RHS[j] # intégrale dans
     le triangle du produit f x fonction de base
    #Contribution de la cellule triangulaire i à la ligne j du syst
    ème linéaire
    for k in [nodeId0,nodeId1,nodeId2] :
      Rigidite.addValue(j,k,GradShapeFuncs[j]*GradShapeFuncs[k]/Ci.
    getMeasure())

print("Linear system matrix building done")

# Conditionnement de la matrice de rigidité
#=================================
cond = Rigidite.getConditionNumber(True)
print("Condition number is ",cond)

# Résolution du système linéaire
#=================================
LS=cdmath.LinearSolver(Rigidite,RHS,100,1.E-6,"GMRES","ILU")
LS.setMatrixIsSingular()#En raison de l'absence de bord
SolSyst=LS.solve()
print("Preconditioner used : ", LS.getNameOfPc() )
print("Number of iterations used : ", LS.getNumberOfIter() )
print("Final residual : ", LS.getResidu() )
print("Linear system solved")

# Création du champ résultat
#===========================
my_ResultField = cdmath.Field("ResultField", cdmath.NODES, my_mesh,
     1)
for j in range(nbNodes):
    my_ResultField[j]=SolSyst[j];#remplissage des valeurs issues du
     système linéaire dans le champs résultat
#sauvegarde sur le disque dur du résultat dans un fichier paraview
my_ResultField.writeVTK("FiniteElementsOnCubeSkinPoisson")
my_RHSfield.writeVTK("RHS_CubeSkinPoisson")

print("Integral of the numerical solution", my_ResultField.integral
    (0))
print("Numerical solution of Poisson equation on a cube skin using
    finite elements done")

#Calcul de l'erreur commise par rapport à la solution exacte
#===========================================================
#The following formulas use the fact that the exact solution is
    equal the right hand side divided by 8*pi*pi
max_abs_sol_exacte=0
erreur_abs=0
max_sol_num=0
min_sol_num=0
for i in range(nbNodes) :
    if max_abs_sol_exacte < abs(my_RHSfield[i]) :
        max_abs_sol_exacte = abs(my_RHSfield[i])
    if erreur_abs  < abs(my_RHSfield[i]/(8*pi*pi) - my_ResultField[
    i]) :
        erreur_abs = abs(my_RHSfield[i]/(8*pi*pi) - my_ResultField[
    i])
    if max_sol_num  < my_ResultField[i] :
        max_sol_num = my_ResultField[i]
    if min_sol_num  > my_ResultField[i] :
        min_sol_num = my_ResultField[i]
max_abs_sol_exacte = max_abs_sol_exacte/(8*pi*pi)

print("Relative error = max(| exact solution - numerical solution
    |)/max(| exact solution |) = ",erreur_abs/max_abs_sol_exacte)
print("Maximum numerical solution = ", max_sol_num, " Minimum
    numerical solution = ", min_sol_num)
print("Maximum exact solution = ", my_RHSfield.max()/(8*pi*pi), "
    Minimum exact solution = ", my_RHSfield.min()/(8*pi*pi) )

#Postprocessing :
#================
# save 3D picture
PV_routines.Save_PV_data_to_picture_file("
    FiniteElementsOnCubeSkinPoisson"+'_0.vtu',"ResultField",'NODES'
    ,"FiniteElementsOnCubeSkinPoisson")
resolution=100
VTK_routines.Clip_VTK_data_to_VTK("FiniteElementsOnCubeSkinPoisson"
    +'_0.vtu',"Clip_VTK_data_to_VTK_"+ "
    FiniteElementsOnCubeSkinPoisson"+'_0.vtu',[0.75,0.75,0.75],
    [0.,0.5,-0.5],resolution )
PV_routines.Save_PV_data_to_picture_file("Clip_VTK_data_to_VTK_"+"
    FiniteElementsOnCubeSkinPoisson"+'_0.vtu',"ResultField",'NODES'
    ,"Clip_VTK_data_to_VTK_"+"FiniteElementsOnCubeSkinPoisson")

# Plot  over slice circle
finiteElementsOnCubeSkin_0vtu = pvs.XMLUnstructuredGridReader(
    FileName=["FiniteElementsOnCubeSkinPoisson"+'_0.vtu'])
slice1 = pvs.Slice(Input=finiteElementsOnCubeSkin_0vtu)
slice1.SliceType.Normal = [0, 1, 0]
renderView1 = pvs.GetActiveViewOrCreate('RenderView')
finiteElementsOnCubeSkin_0vtuDisplay = pvs.Show(
    finiteElementsOnCubeSkin_0vtu, renderView1)
pvs.ColorBy(finiteElementsOnCubeSkin_0vtuDisplay, ('POINTS', '
    ResultField'))
slice1Display = pvs.Show(slice1, renderView1)
pvs.SaveScreenshot("./FiniteElementsOnCubeSkinPoisson"+"_Slice"+'.
    png', magnification=1, quality=100, view=renderView1)
plotOnSortedLines1 = pvs.PlotOnSortedLines(Input=slice1)
lineChartView2 = pvs.CreateView('XYChartView')
plotOnSortedLines1Display = pvs.Show(plotOnSortedLines1,
    lineChartView2)
plotOnSortedLines1Display.UseIndexForXAxis = 0
plotOnSortedLines1Display.XArrayName = 'arc_length'
plotOnSortedLines1Display.SeriesVisibility = ['ResultField (1)']
pvs.SaveScreenshot("./FiniteElementsOnCubeSkinPoisson"+"
    _PlotOnSortedLine_"+'.png', magnification=1, quality=100, view=
    lineChartView2)
pvs.Delete(lineChartView2)

assert erreur_abs/max_abs_sol_exacte <1.
```

References

[1] J. Lafontaine, An introduction to differentiable manifolds, Springer, 2015

[2] E. Hebey, Nonlinear analysis on manifolds ; Sobolev spaces and inequalities, *Courant Lecture Notes*

[3] F. Gesztesy, I. Mitrea, D. Mitrea, M. Mitrea, On the nature of the Laplace-Beltrami operator on lipschitz manifolds, Journal of Mathematical Sciences, Vol. **172**(2011), No. 3, pages 279–346.

[4] V. Gol'dshtein, I. Mitrea, M. Mitrea, Hodge Decompositions with mixed boundary conditions and application to partial differential equations on lipschitz manifolds, Journal of Mathematical Sciences, Vol. **172**(2011), No. 3, pages 347–400.

[5] J. Luukkainen, J. Väisälä, Elements of Lipschitz Topology, Annales Academire Scientiarum Fennicre, Series A. I. Mathematica, Volumen **3**(1977), 85-122.

[6] G. Dziuk, Finite elements for the Beltrami operator on arbitrary surfaces. *in Partial differential equations and calculus of variations, S. Hildebrandt and R. Leis, eds., vol. 1357 of Lecture Notes in Mathematics, Springer*,1988, pp. 142-155.

[7] G. Dziuk and C. M. Elliott, Finite element methods for surface PDEs, *Acta Numerica* 2013, pp. 289-396

[8] Y. Saad. Iterative Methods for Sparse Linear Systems.*PWS Publishing Company, Boston*, 1996.

[9] M. Ndjinga, M. Nguemfouo, On the finite element method for the Laplace-Beltrami operator on closed Lipschitz manifolds, submitted

[10] M. Ndjinga, M. Nguemfouo, On the condition number of the nite element method for Laplace-Beltrami operator on closed surfaces, submitted

[11] Ribes, Andre, and Christian Caremoli. "Salome platform component model for numerical simulation." Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International. Vol. 2. IEEE,2007.

[12] http://www.salome-platform.org/downloads/current-version

[13] https://github.com/ndjinga/SOLVERLAB