

# Simulation of the Poisson problem on the 3D Sphere

Marcial Nguemfouo, PhD Student

May 2, 2020

## 1 Introduction

We consider the unit sphere defined as follows

$$\Gamma = \{(x, y, z) \in \mathbb{R}^3, x^2 + y^2 + z^2 = 1\}.$$

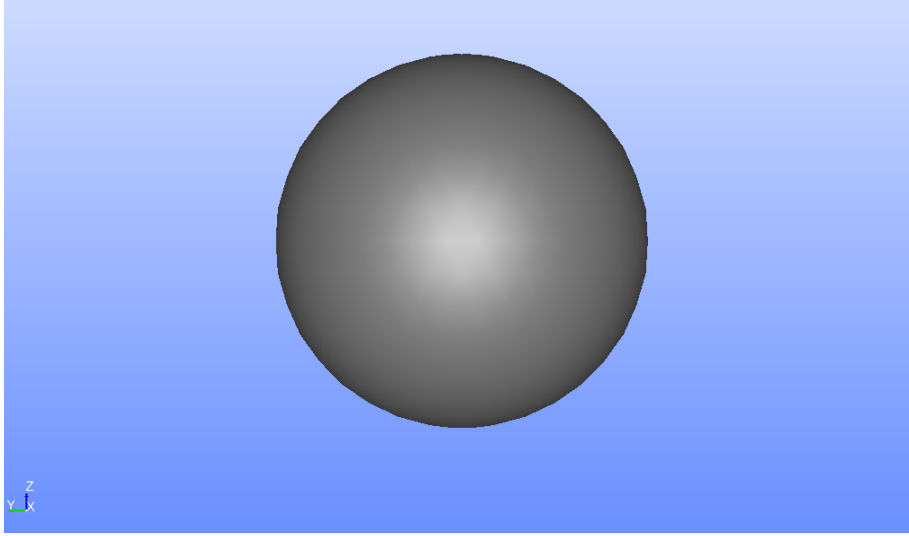


Figure 1: The unit sphere in SALOME CAO module

The sphere is a  $C^\infty$  manifold where the **Laplace-Beltrami operator**  $\Delta_\Gamma$ , a generalisation of the euclidean laplacean, can be defined as the combination of a surface divergence  $\nabla_\Gamma \cdot$ , and of a surface gradient  $\vec{\nabla}_\Gamma$  (see for instance [3, 6, 12]).

Using the spherical coordinates  $(\theta, \phi)$  where  $\theta$  is the longitude and  $\phi$  the latitude, the Laplace-Beltrami operator takes the following form on the sphere

$$\Delta_\Gamma f = \frac{1}{\sin(\phi)} \frac{\partial}{\partial \phi} \left( \sin \phi \frac{\partial f}{\partial \phi} \right) + \frac{1}{\sin(\phi)^2} \frac{\partial^2 f}{\partial \theta^2}.$$

We consider the following **Poisson problem** on the sphere

$$\begin{cases} -\Delta_\Gamma u = f \text{ on } \Gamma \\ \int_\Gamma u = 0 \end{cases}, \quad (1)$$

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

For the following choice of  $f$  :

$$f(x, y, z) = \frac{12}{(x^2 + y^2 + z^2)^{\frac{3}{2}}} (3x^2y - y^3),$$

the exact solution  $u$  of (1) is given by (see [5]):

$$u(x, y, z) = \frac{1}{(x^2 + y^2 + z^2)^{\frac{3}{2}}} (3x^2y - y^3) = \frac{1}{12} f.$$

Our objective is to solve numerically the Poisson problem (1) using the finite element method described in [1, 2, 3, 4].

## 2 Finite elements method for 3D Poisson problem

Since  $\Gamma$  is closed (no boundary), we have to impose the global condition  $\int_\Gamma u = 0$  to guarantee the uniqueness of solution. For this reason we define the following Lebesgue space

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

and the following Sobolev space

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

### 2.1 Well-posedness of the problem

#### 2.1.1 Variational formulation and Poincaré inequality

In this section we are going to recall two important properties of the surface gradient operator  $\vec{\nabla}_\Gamma$  : the Green-Ostrogradski formula (i.e. integration by part) and Poincaré inequality.

Thanks to the **Green-Ostrogradski** theorem, the variational formulation of (1) is:

$$\text{Find } u \in H^1_\#(\Gamma) \text{ such that } \forall v \in H^1_\#(\Gamma), \int_\Gamma \vec{\nabla}_\Gamma u \cdot \vec{\nabla}_\Gamma v = \int_\Gamma f v. \quad (2)$$

As for the classical gradient, there is a **Poincaré's inequality** involving the surface gradient (see theorem 2.12 in [3]).

**Theorem 1** (Poincaré's inequality).

*Assume that  $\Gamma$  is an embedded  $C^3$  hypersurface. There exists a constant  $c$  such that, for every function  $f \in H^1(\Gamma)$  with  $\int_\Gamma f = 0$ , we have the inequality*

$$\|f\|_{L^2(\Gamma)} \leq c \|\nabla_\Gamma f\|_{L^2(\Gamma)}.$$

### 2.1.2 Existence of a unique weak solution

The bilinear form

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

is continuous and coercive thanks to Poincaré inequality.

The linear form

$$b(v) = \int_{\Gamma} f v$$

is continuous.

By application of the **Lax-Milgram theorem**, the variational formulation (2) of problem (1) admits a unique weak solution, which depends continuously on the data  $f$  (see Theorem 3.1 in [3]).

**Theorem 2** (Well-posedness). *Let  $\Gamma \in C^2$ , be a compact hypersurface in  $\mathbb{R}^3$  and assume that  $f \in L^2(\Gamma)$  and  $\int_{\Gamma} f = 0$ . Then there exists a unique solution  $u \in H^1(\Gamma)$  of (2) with  $\int_{\Gamma} u = 0$ .*

### 2.1.3 Regularity of the solution

The regularity of the solution requires the regularity of both the right hand side and the manifold. The following theorem is taken from [3] theorem 3.3.

**Theorem 3.** *Take an integer  $m \geq 0$ . Let  $\Gamma$  be an compact manifold of class  $C^{m+2}$ . Let  $f \in H_{\#}^m(\Gamma)$ . Then, the unique solution  $u_f \in H_{\#}^1(\Gamma)$  of (2) belongs to  $H^{m+2}(\Gamma)$ . Further, the mapping  $f \rightarrow u_f$  is linear and continuous from  $H_{\#}^m(\Gamma)$  into  $H_{\#}^{m+2}(\Gamma)$ , that is, there exists a constant  $C > 0$  such that*

$$\|u_f\|_{H_{\#}^{m+2}(\Gamma)} \leq C \|f\|_{H_{\#}^m(\Gamma)}.$$

For more details see [1] for euclidian case and [2, 3] for the case of curved surfaces.

## 2.2 The P1 finite elements

Following [3], we first approximate the sphere  $\Gamma$  by a polyhedral surface  $\Gamma_h$  with triangular faces  $(\mathcal{T}_k)_{k \geq 1}$  called elements having their nodes on  $\Gamma$ . We approximate functions  $f \in H_{\#}^1(\Gamma)$  by functions  $f_h \in H_{\#}^1(\Gamma_h)$  via the lift operator (10).

We consider  $u_h$  the weak solution of the following Poisson problem on the piecewise linear manifold  $\Gamma_h$  :

$$\begin{cases} -\Delta_{\Gamma_h} u_h = f_h \text{ on } \Gamma_h \\ u_h \in H_{\#}^1(\Gamma_h) \end{cases}, \quad (3)$$

and its variational formulation, analog to (2) is

$$\text{Find } u_h \in H_{\#}^1(\Gamma_h) \text{ such that } \forall v_h \in H_{\#}^1(\Gamma_h), \int_{\Gamma_h} \vec{\nabla}_{\Gamma_h} u_h \cdot \vec{\nabla}_{\Gamma_h} v_h = \int_{\Gamma_h} f_h v_h. \quad (4)$$

We look for  $\tilde{u}_h$  the projection of the solution  $u_h$  of (4) on the space  $V_0(\Gamma_h)$  of continuous piecewise affine functions with zero mean on  $\Gamma_h$ .

The discrete form of the variational formulation (4) is then given by.

$$\text{Find } \tilde{u}_h \in V_0(\Gamma_h) \text{ such that } \forall \tilde{v}_h \in V_0(\Gamma_h), \int_{\Gamma_h} \vec{\nabla}_{\Gamma_h} \tilde{u}_h \cdot \vec{\nabla}_{\Gamma_h} \tilde{v}_h = \int_{\Gamma_h} \tilde{f}_h \tilde{v}_h, \quad (5)$$

where  $\tilde{f}_h$  is the projection of  $f_h$  on  $V_0(\Gamma_h)$ .

### 2.3 The linear system to be solved

Since  $V_0(\Gamma_h)$  is generated by the nodal functions  $\phi_i : \Gamma_h \rightarrow \mathbb{R}$ ,  $i = 1, \dots, n$  such that  $\phi_i(x_j) = \delta_{ij}$ , (5) takes the following algebraic form

$$A_{\Delta_{\Gamma_h}} X = b_h, \quad (6)$$

where

$$\tilde{u}_h = \sum_{i=1}^n u_i \phi_i, \quad (7)$$

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

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

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

$A_{\Delta_{\Gamma_h}}$  is symmetric positive and sparse but not invertible since constants are in its kernel, hence the linear system (6) is singular. However it admits a unique solution with zero mean provided the right hand side has zero mean (see [3]).

### 2.4 Convergence of the numerical method

#### 2.4.1 Fermi coordinates and lift operator

A function  $u$  defined on  $\Gamma$  can be extended to a neighborhood of  $\Gamma$  in  $\mathbb{R}^3$  using a lift operator based on the Fermi coordinates around  $\Gamma$ . Following [3], we define the  **$\delta$ -strip around  $\Gamma$**  as

$$U_{\delta,\Gamma} = \{x \in \mathbb{R}^3, \text{dist}(x, \Gamma) < \delta\}. \quad (8)$$

For  $\delta$  small enough it is possible to define the projection  $a : U_{\delta} \rightarrow \Gamma$  onto  $\Gamma$  and the distance function  $d : U_{\delta} \rightarrow \mathbb{R}_+$  to  $\Gamma$ .  $a(x)$  and  $d(x)$  are called the **Fermi coordinates** of  $x$  and their existence is given by the following theorem (see Lemma 2.8 in [3] for the proof).

**Theorem 4** (Fermi coordinates).

*Let  $\Gamma$  be an embedded  $C^2$  hypersurface. There exists  $\delta_{Fermi} > 0$  such that for every point  $x \in U_{\delta_{Fermi},\Gamma}$ , there exists a unique point  $a(x) \in \Gamma$ , and a function  $d \in C^2(U_{\delta_{Fermi},\Gamma})$  such that*

$$\forall x \in U_{\delta_{Fermi},\Gamma}, \quad x = a(x) + d(x) \vec{n}(x), \quad (9)$$

where  $\vec{n}(x)$  is the unit normal vector to  $\Gamma$  at  $x$ .

Thanks to the **Fermi coordinates** defined in theorem 4, we can define as in [3] (equation 4.2) a **lift operator**  $L$  such that

$$\begin{aligned} L : C(\Gamma_h) &\rightarrow C(\Gamma) \\ u_h &\rightarrow u_h \circ a^{-1} \end{aligned} \quad (10)$$

provided

$$\Gamma_h \subset U_{\delta_{Fermi}, \Gamma}. \quad (11)$$

### 2.4.2 Convergence theorems

In order to study the convergence of the finite element approximation, we need to compare  $u \in H^1(\Gamma)$  with  $\tilde{u}_h \in H^1(\Gamma_h)$  but don't share the same support. Hence we need to use the lift operator (10) which requires the assumption (11) that the triangulated surface  $\Gamma_h$  is close enough to  $\Gamma$ .

As the parameter  $h$  goes to zero the distance between  $\Gamma_h$  and  $\Gamma$  converges to zero as expressed in the following theorem taken from [3] Lemma 4.1.

**Theorem 5** (Convergence of  $\Gamma_h$  towards  $\Gamma$ ). *Let  $\Gamma \in \mathbb{R}^3$  be an embedded  $C^2$  hypersurface and  $\Gamma_h \subset U_{\delta_{Fermi}, \Gamma}$  a piecewise linear surface. Let  $h$  be the largest diameter of triangles in  $\Gamma_h$ . There exists a constant  $c$  such that*

$$\forall x \in \Gamma_h, \quad dist(x, \Gamma) \leq ch^2.$$

Once proven that  $\Gamma_h$  converges towards  $\Gamma$ , we can prove that  $\tilde{u}_h$  converges to  $u$  using the lift operator (10). The following convergence theorem is taken from [2] Theorem 8, Lemma 6 and Lemma 7.

**Theorem 6** (Convergence of  $\tilde{u}_h$  towards  $u$ ). *Let  $\Gamma \in \mathbb{R}^3$  be an embedded  $C^2$  hypersurface and  $\Gamma_h \subset U_{\delta_{Fermi}, \Gamma}$  a piecewise linear surface. Let  $h$  be the largest diameter of triangles in  $\Gamma_h$ .*

*If  $u$  is a continuous solution of the Poisson problem (1) and  $\tilde{u}_h$  is the discrete solution of (5), then there exists  $c > 0$  such that*

$$\|u - \tilde{u}_h \circ a^{-1}\|_{L^2(\Gamma)} \leq ch^2, \quad \|\nabla_\Gamma(u - \tilde{u}_h \circ a^{-1})\|_{L^2(\Gamma)} \leq ch. \quad (12)$$

## 3 Numerical results for Laplace-Beltrami operator on Sphere

For the coding the finite element method, we use the Python language and the open-source Linux based library CDMATH [11] which is very simple for the manipulation of large matrices, vectors, meshes and fields. It (CDMATH) can handle finite element and finite volume discretizations, read general 3D geometries and meshes generated by SALOME.

### 3.1 Meshing of the domain

For the design and meshing of the domain we use GEOMETRY and MESH modules of the software SALOME 9.5 (see [8, 10, 9]).

Below are the meshes used in our convergence analysis.

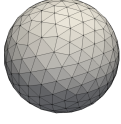
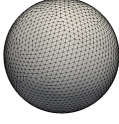
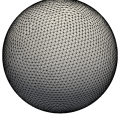
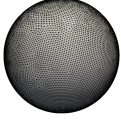
meshSphere 1	meshSphere 2	meshSphere 3	meshSphere 4
			
288 cells	2638 cells	4512 cells	10773 cells

Figure 2: Mesh of domain

### 3.2 Visualization of the results

For the numerical resolution of our discrete problem, we use an iterative solver because the stiffness matrix  $A_{\Delta_{\Gamma_h}}$  is large and sparse (see [7]) .

For the visualization of the result, we use the PARAVIS module included in SALOME (see [9]).

Below are visualizations of the numerical results obtained on the different meshes of picture 2.

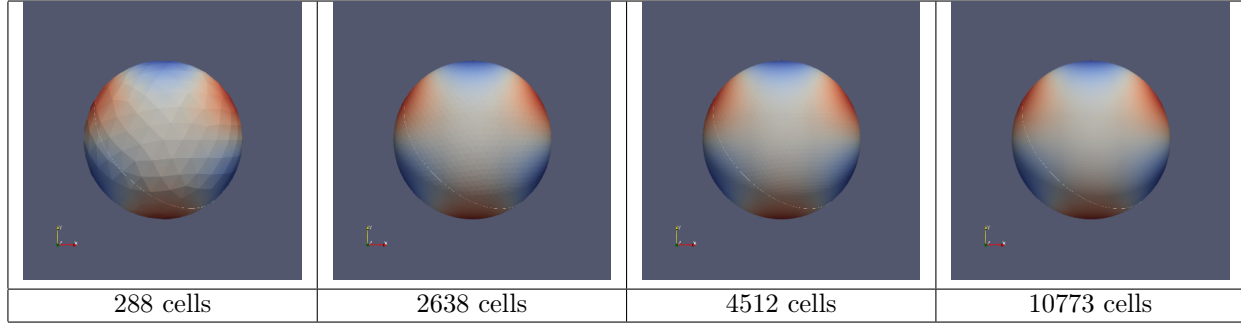


Figure 3: Numerical results of the finite elements on the unit sphere

Below are clipings of the previous numerical results.

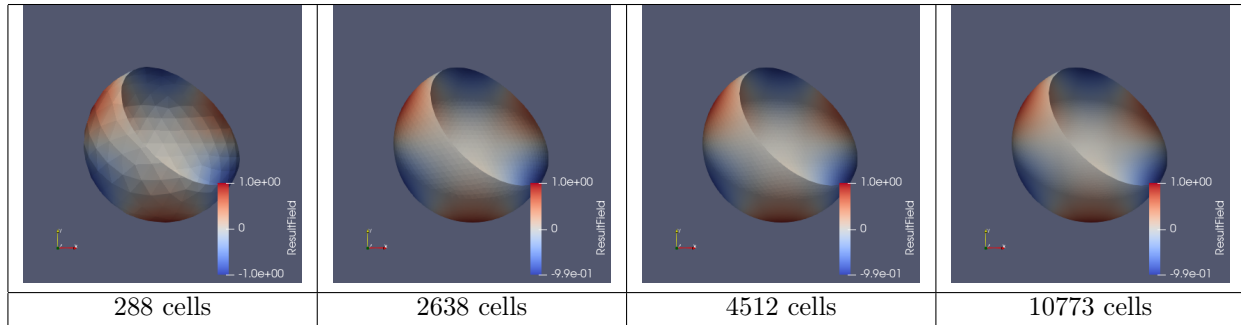


Figure 4: Clipping of the numerical result on the unit sphere

### 3.3 Numerical convergence of the finite element method

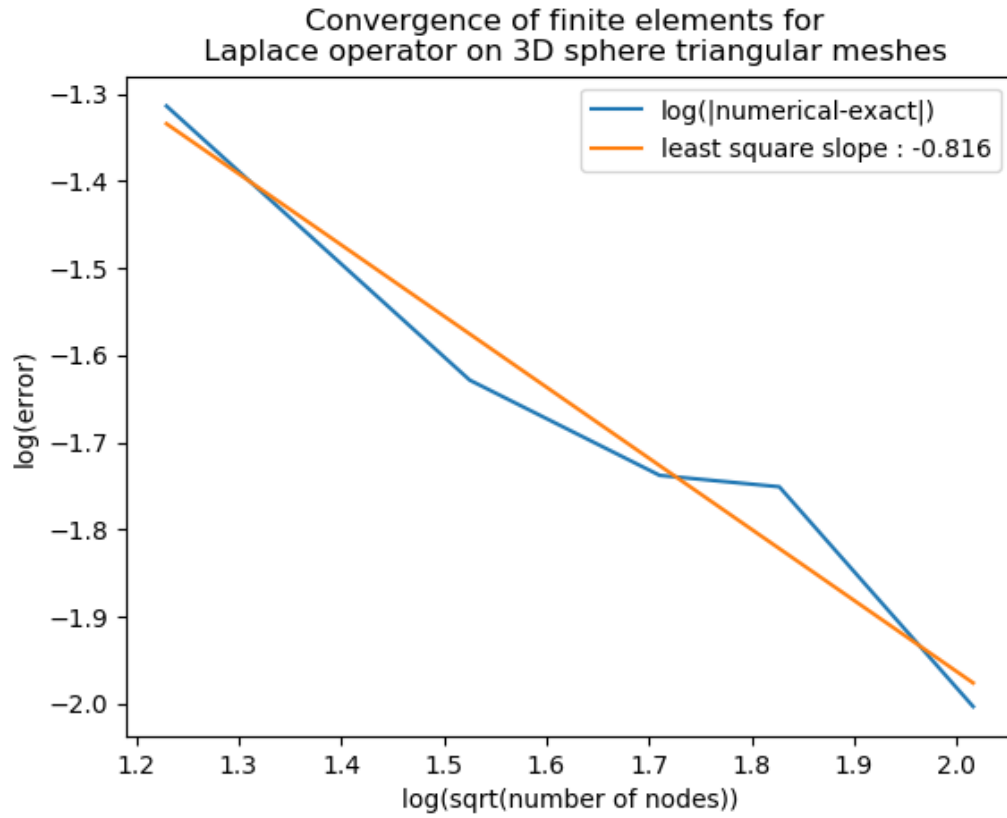


Figure 5: Convergence of the finite element method on the sphere

The method converges with a numerical order of approximately 0.8.

### 3.4 Computational time of the finite element method

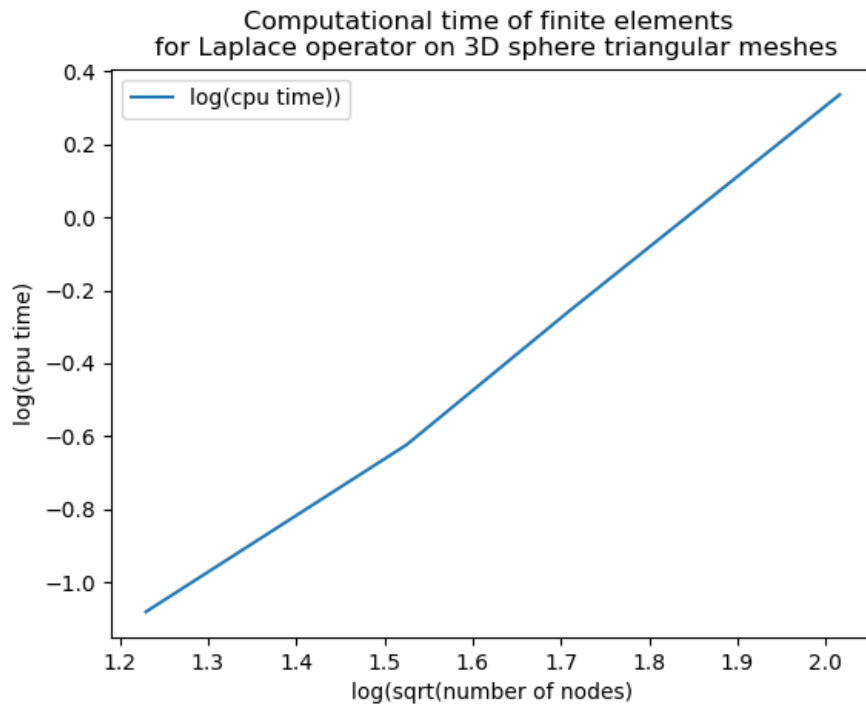


Figure 6: Computational time of the finite element method on the sphere



### 3.5 Ploting over slice circle

Here we have drawn a circle on each sphere to extract the values. This circle is visible on the spheres in Figure 3.

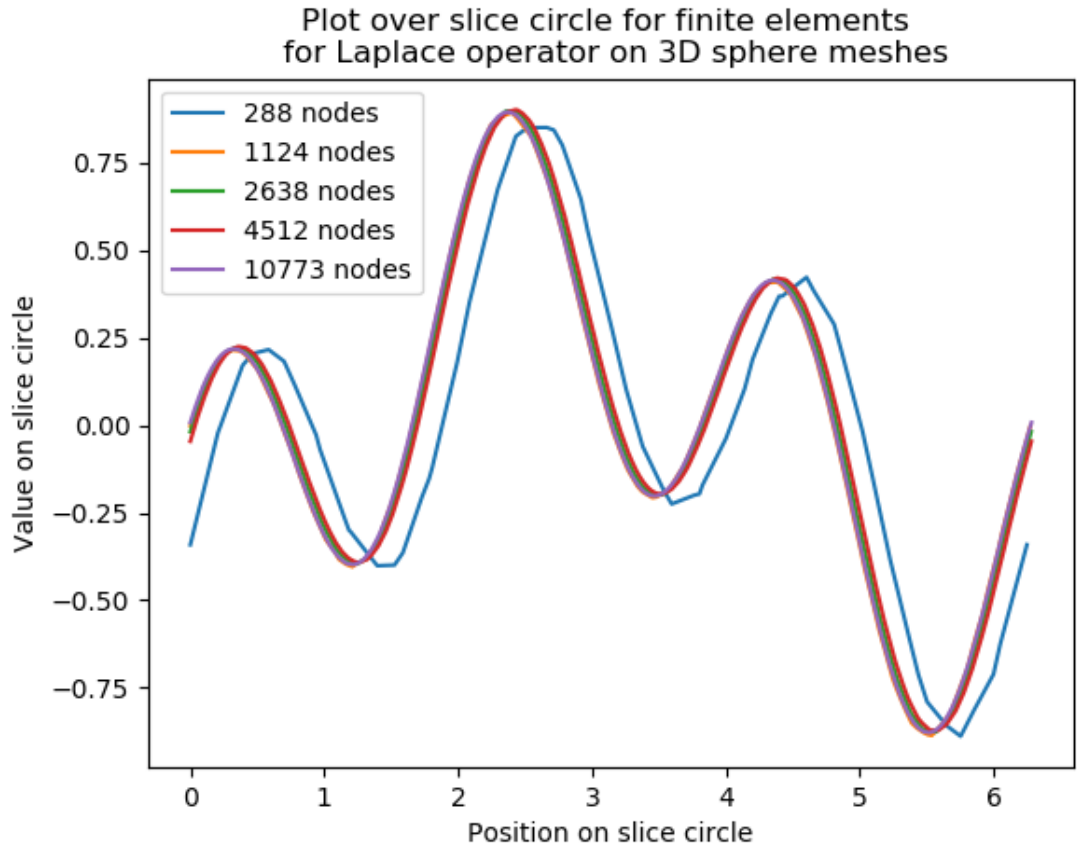


Figure 7: Convergence of the data plotted over a circle drawn on the sphere

## 4 The script

```
1 # -*- coding: utf-8 -*-
2 #
3 # Name      : Résolution EF de l'équation de Laplace-Beltrami -\
4 #            triangle u = f sur une sphere
5 # Author    : Michael Ndjinga
6 # Copyright : CEA Saclay 2017
7 # Description : Utilisation de la méthode des éléments finis P1
8 #              avec champs u et f discrétisés aux noeuds d'un maillage
9 #              triangulaire
10 #
11 #           Création et sauvegarde du champ résultant ainsi que
12 #           du champ second membre en utilisant la librairie CDMATH
```

```

8 # Référence : M. A. Olshanskii, A. Reusken, and J.
  Grande. A finite element method for elliptic equations
9 # on surfaces. SIAM J. Num. Anal., 47, p.
  3355
10 # Solution exacte = f/12 : il s'agit d'un vecteur
  propre du laplacien sur la sphère
11 # Résolution d'un système linéaire à matrice singuliè
  re : les vecteurs constants sont dans le noyau
12 #
  =====

13
14 import cdmath
15 from math import pow
16 import numpy as np
17 import PV_routines
18 import VTK_routines
19 import paraview.simple as pvs
20
21 #Chargement du maillage triangulaire de la sphère
22 #
  =====

23 my_mesh = cdmath.Mesh("meshSphere.med")
24 if(not my_mesh.isTriangular()) :
25     raise ValueError("Wrong cell types : mesh is not made of
  triangles")
26 if(my_mesh.getMeshDimension()!=2) :
27     raise ValueError("Wrong mesh dimension : expected a surface of
  dimension 2")
28 if(my_mesh.getSpaceDimension()!=3) :
29     raise ValueError("Wrong space dimension : expected a space of
  dimension 3")
30
31 nbNodes = my_mesh.getNumberOfNodes()
32 nbCells = my_mesh.getNumberOfCells()
33
34 print("Mesh building/loading done")
35 print("nb of nodes=", nbNodes)
36 print("nb of cells=", nbCells)
37
38 #Discrétisation du second membre et détermination des noeuds inté
  rieurs
39 #
  =====

40 my_RHSfield = cdmath.Field("RHS field", cdmath.NODES, my_mesh, 1)
41 maxNbNeighbours = 0#This is to determine the number of non zero
  coefficients in the sparse finite element rigidity matrix
42
43 #parcours des noeuds pour discrétisation du second membre et
  extraction du nb max voisins d'un noeud
44 for i in range(nbNodes):
45     Ni=my_mesh.getNode(i)
46     x = Ni.x()
47     y = Ni.y()
48     z = Ni.z()
49
50     my_RHSfield[i]=12*y*(3*x*x-y*y)/pow(x*x+y*y+z*z,3/2)#vecteur
  propre du laplacien sur la sphère
51     if my_mesh.isBorderNode(i): # Détection des noeuds frontière
52         raise ValueError("Mesh should not contain borders")

```

```

53     else:
54         maxNbNeighbours = max(1+Ni.getNumberOfCells(),maxNbNeighbours)
55         #true only for planar cells, otherwise use function Ni.
56         getNumberOfEdges()
57
58     print("Right hand side discretisation done")
59     print("Max nb of neighbours=", maxNbNeighbours)
60     print("Integral of the RHS", my_RHSfield.integral(0))
61
62     # Construction de la matrice de rigidité et du vecteur second
63     # membre du système linéaire
64
65     =====
66
67     Rigidite=cdmath.SparseMatrixPetsc(nbNodes,nbNodes,maxNbNeighbours)#
68         warning : third argument is number of non zero coefficients
69         per line
70     RHS=cdmath.Vector(nbNodes)
71
72     # Vecteurs gradient de la fonction de forme associée à chaque noeud
73     # d'un triangle
74     GradShapeFunc0=cdmath.Vector(3)
75     GradShapeFunc1=cdmath.Vector(3)
76     GradShapeFunc2=cdmath.Vector(3)
77
78     normalFace0=cdmath.Vector(3)
79     normalFace1=cdmath.Vector(3)
80
81     #On parcourt les triangles du domaine
82     for i in range(nbCells):
83
84         Ci=my_mesh.getCell(i)
85
86         #Contribution à la matrice de rigidité
87         nodeId0=Ci.getNodeId(0)
88         nodeId1=Ci.getNodeId(1)
89         nodeId2=Ci.getNodeId(2)
90         N0=my_mesh.getNode(nodeId0)
91         N1=my_mesh.getNode(nodeId1)
92         N2=my_mesh.getNode(nodeId2)
93
94         #Build normal to cell Ci
95         normalFace0[0]=Ci.getNormalVector(0,0)
96         normalFace0[1]=Ci.getNormalVector(0,1)
97         normalFace0[2]=Ci.getNormalVector(0,2)
98         normalFace1[0]=Ci.getNormalVector(1,0)
99         normalFace1[1]=Ci.getNormalVector(1,1)
100        normalFace1[2]=Ci.getNormalVector(1,2)
101
102        normalCell = normalFace0.crossProduct(normalFace1)
103        normalCell = normalCell/normalCell.norm()
104
105        cellMat=cdmath.Matrix(4)
106        cellMat[0,0]=N0.x()
107        cellMat[0,1]=N0.y()
108        cellMat[0,2]=N0.z()
109        cellMat[1,0]=N1.x()
110        cellMat[1,1]=N1.y()
111        cellMat[1,2]=N1.z()
112        cellMat[2,0]=N2.x()
113        cellMat[2,1]=N2.y()
114        cellMat[2,2]=N2.z()

```

```

107 cellMat[3,0]=normalCell[0]
108 cellMat[3,1]=normalCell[1]
109 cellMat[3,2]=normalCell[2]
110 cellMat[0,3]=1
111 cellMat[1,3]=1
112 cellMat[2,3]=1
113 cellMat[3,3]=0
114
115 #Formule des gradients voir EF P1 -> calcul déterminants
116 GradShapeFunc0[0]= cellMat.partMatrix(0,0).determinant()/2
117 GradShapeFunc0[1]=-cellMat.partMatrix(0,1).determinant()/2
118 GradShapeFunc0[2]= cellMat.partMatrix(0,2).determinant()/2
119 GradShapeFunc1[0]=-cellMat.partMatrix(1,0).determinant()/2
120 GradShapeFunc1[1]= cellMat.partMatrix(1,1).determinant()/2
121 GradShapeFunc1[2]=-cellMat.partMatrix(1,2).determinant()/2
122 GradShapeFunc2[0]= cellMat.partMatrix(2,0).determinant()/2
123 GradShapeFunc2[1]=-cellMat.partMatrix(2,1).determinant()/2
124 GradShapeFunc2[2]= cellMat.partMatrix(2,2).determinant()/2
125
126 #Création d'un tableau (numéro du noeud, gradient de la fonction
    de forme
127 GradShapeFuncs={nodeId0 : GradShapeFunc0}
128 GradShapeFuncs[nodeId1]=GradShapeFunc1
129 GradShapeFuncs[nodeId2]=GradShapeFunc2
130
131 # Remplissage de la matrice de rigidité et du second membre
132 for j in [nodeId0,nodeId1,nodeId2] :
133     #Ajout de la contribution de la cellule triangulaire i au
    second membre du noeud j
134     RHS[j]=Ci.getMeasure()/3*my_RHSfield[j]+RHS[j] # intégrale dans
    le triangle du produit f x fonction de base
135     #Contribution de la cellule triangulaire i à la ligne j du syst
    ème linéaire
136     for k in [nodeId0,nodeId1,nodeId2] :
137         Rigidite.addValue(j,k,GradShapeFuncs[j]*GradShapeFuncs[k]/Ci.
            getMeasure())
138
139 print("Linear system matrix building done")
140
141 # Résolution du système linéaire
142 #=====
143 LS=cdmath.LinearSolver(Rigidite,RHS,100,1.E-6,"GMRES","ILU")
144 LS.setMatrixIsSingular()#En raison de l'absence de bord
145 SolSyst=LS.solve()
146 print "Preconditioner used : ", LS.getNameOfPc()
147 print "Number of iterations used : ", LS.getNumberOfIter()
148 print "Final residual : ", LS.getResidu()
149 print("Linear system solved")
150
151 # Création du champ résultat
152 #=====
153 my_ResultField = cdmath.Field("ResultField", cdmath.NODES, my_mesh,
    1)
154 for j in range(nbNodes):
155     my_ResultField[j]=SolSyst[j];#remplissage des valeurs pour les
    noeuds intérieurs
156 #sauvegarde sur le disque dur du résultat dans un fichier paraview
157 my_ResultField.writeVTK("FiniteElementsOnSpherePoisson")
158
159 #Postprocessing :
160 #=====
161 # save 3D picture

```

```

162 PV_routines.Save_PV_data_to_picture_file("
    FiniteElementsOnSpherePoisson"+'_0.vtu',"ResultField",'NODES',"
    FiniteElementsOnSpherePoisson")
163 resolution=100
164 VTK_routines.Clip_VTK_data_to_VTK("FiniteElementsOnSpherePoisson"+
    '_0.vtu',"Clip_VTK_data_to_VTK_"+ "FiniteElementsOnSpherePoisson"
    "+'_0.vtu',[0.25,0.25,0.25], [-0.5,-0.5,-0.5],resolution )
165 PV_routines.Save_PV_data_to_picture_file("Clip_VTK_data_to_VTK_"+
    "FiniteElementsOnSpherePoisson"+'_0.vtu',"ResultField",'NODES',"
    Clip_VTK_data_to_VTK_"+ "FiniteElementsOnSpherePoisson")
166
167 # Plot over slice circle
168 finiteElementsOnSphere_Ovtu = pvs.XMLUnstructuredGridReader(
    FileName=["FiniteElementsOnSpherePoisson"+'_0.vtu'])
169 slice1 = pvs.Slice(Input=finiteElementsOnSphere_Ovtu)
170 slice1.SliceType.Normal = [0.5, 0.5, 0.5]
171 renderView1 = pvs.GetActiveViewOrCreate('RenderView')
172 finiteElementsOnSphere_OvtuDisplay = pvs.Show(
    finiteElementsOnSphere_Ovtu, renderView1)
173 pvs.ColorBy(finiteElementsOnSphere_OvtuDisplay, ('POINTS', '
    ResultField'))
174 slice1Display = pvs.Show(slice1, renderView1)
175 pvs.SaveScreenshot("./FiniteElementsOnSpherePoisson"+ "_Slice"+'.png'
    , magnification=1, quality=100, view=renderView1)
176 plotOnSortedLines1 = pvs.PlotOnSortedLines(Input=slice1)
177 lineChartView2 = pvs.CreateView('XYChartView')
178 plotOnSortedLines1Display = pvs.Show(plotOnSortedLines1,
    lineChartView2)
179 plotOnSortedLines1Display.UseIndexForXAxis = 0
180 plotOnSortedLines1Display.XArrayName = 'arc_length'
181 plotOnSortedLines1Display.SeriesVisibility = ['ResultField (1)']
182 pvs.SaveScreenshot("./FiniteElementsOnSpherePoisson"+
    "_PlotOnSortedLine_"+'.png', magnification=1, quality=100, view=
    lineChartView2)
183 pvs.Delete(lineChartView2)
184
185 print("Integral of the numerical solution", my_ResultField.integral
    (0))
186 print("Numerical solution of Poisson equation on a sphere using
    finite elements done")
187
188 #Calcul de l'erreur commise par rapport à la solution exacte
189 #=====
190 #The following formulas use the fact that the exact solution is
    equal the right hand side divided by 12
191 max_abs_sol_exacte=0
192 erreur_abs=0
193 max_sol_num=0
194 min_sol_num=0
195 for i in range(nbNodes) :
196     if max_abs_sol_exacte < abs(my_RHSfield[i]) :
197         max_abs_sol_exacte = abs(my_RHSfield[i])
198     if erreur_abs < abs(my_RHSfield[i]/12 - my_ResultField[i]) :
199         erreur_abs = abs(my_RHSfield[i]/12 - my_ResultField[i])
200     if max_sol_num < my_ResultField[i] :
201         max_sol_num = my_ResultField[i]
202     if min_sol_num > my_ResultField[i] :
203         min_sol_num = my_ResultField[i]
204 max_abs_sol_exacte = max_abs_sol_exacte/12
205
206 print("Absolute error = max(| exact solution - numerical solution
    |) = ",erreur_abs )

```

```

207 print("Relative error = max(| exact solution - numerical solution
    |)/max(| exact solution |) = ", erreur_abs/max_abs_sol_exacte)
208 print("Maximum numerical solution = ", max_sol_num, " Minimum
    numerical solution = ", min_sol_num)
209 print("Maximum exact solution = ", my_RHSfield.max()/12, " Minimum
    exact solution = ", my_RHSfield.min()/12 )
210
211 assert erreur_abs/max_abs_sol_exacte <1.

```

## References

- [1] G. Allaire, Numerical analysis and optimization - an introduction to mathematical modeling and numerical simulation, *Oxford University Press*, 2007.
- [2] 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.
- [3] G. Dziuk and C. M. Elliott, Finite element methods for surface PDEs, *Acta Numerica* 2013, pp. 289-396
- [4] A. Ern, and J-L. Guermond, Theory and practice of finite elements. Vol. 159. Springer Science & Business Media, 2013.
- [5] M. A. Olshanskii, A. Reusken and J. Grande, A finite element method for elliptic equations on surfaces, *SIAM J. Numer. Anal.* **47**(2009) 3339-3358.
- [6] L. Qin, S. Zhang and Z. Zhang, Finite Element formulation in flat coordinate spaces to solve elliptic problems in general Riemannian manifolds, *SIAM J. S CI.COMPUT* Vol. **36**(2014) No. 5, pp. A2149-A2165.
- [7] Y. Saad. Iterative Methods for Sparse Linear Systems. *PWS Publishing Company, Boston*, 1996.
- [8] 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.
- [9] <http://www.salome-platform.org/downloads/current-version>, accessed August 2017
- [10] A. Ribes, A. Bruneton, A. Geay, SALOME: an Open-Source simulation platform integrating ParaView, 10.13140/RG.2.2.12107.08485, 2017
- [11] <https://github.com/ndjinga/CDMATH>
- [12] C. Yaiza, *Analysis on Manifolds via the Laplacian*, Math 253, Fall, Harvard University, 2013