Simulation of the Poisson problem on the 3D Torus

Marcial Nguemfouo, PhD Student

May 3, 2020

1 Introduction

We consider the torus defined as follow

$$\Gamma = \{(x, y, z) \in \mathbb{R}^3, (\sqrt{x^2 + y^2} - R)^2 + z^2 - r^2 = 0\},$$

where $r > 0$ is the minor radius and $R > r$ is the major radius.

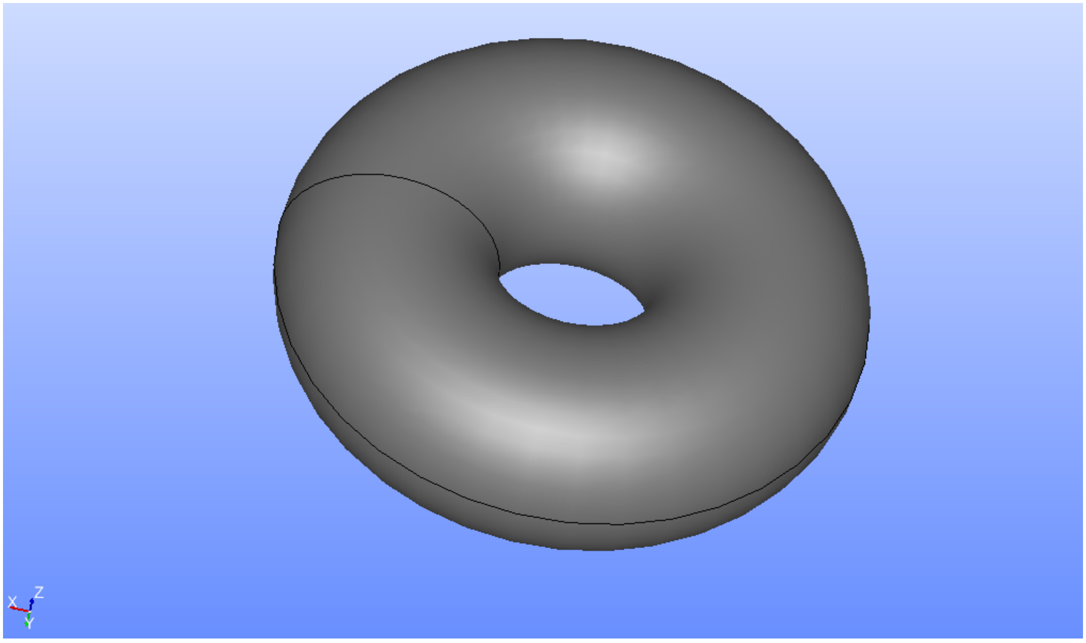


Figure 1: The torus in SALOME CAO module

The torus is a C^∞ manifold of dimension 2 embedded in $\mathbb{R}^3$. The **Laplace-Beltrami operator** $\triangle_\Gamma$, a generalisation of the euclidean laplacean, can be defined on the torus 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 torus coordinates (θ, ϕ) where $\theta, \phi \in [0, 2\pi]$, the Laplace-Beltrami operator takes the following form on the torus

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

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

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

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 $u(\theta, \phi) = \sin(3\phi)\cos(3\theta + \phi)$ (see [5]), the right hand side is given by

$$\begin{aligned} f \quad = \quad & 9r^{-2}\sin(3\phi)\cos(3\theta+\phi) - (10\sin(3\phi)\cos(3\theta+\phi) \\ & +6\cos(3\phi)\sin(3\theta+\phi))((R + rcos(\theta))^{-2} \\ & -3\sin(\theta)\sin(3\phi)\sin(3\theta+\phi)(r(R + r\cos(\theta)))^{-1}. \end{aligned}$$

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

2 Finite elements method for 3D Poisson problem

Since Γ 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-Ostrograski** theorem, the variational formulation of (1) is:

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

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 Γ *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

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

is continuous and coercive thanks to Poincaré inequality.
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, which depends continuously on the data f (see Theorem 3.1 in [3]).

Theorem 2 (Well-posedness). *Let* Γ*, be a compact* C^2 *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. *Let* Γ*, be a compact* C^2 *hypersurface in* $\mathbb{R}^3$ *and assume that* $f \in L^2(\Gamma)$ *and* $\int_\Gamma f = 0$*. Then, the unique weak solution of* (2) *satisfies* $u \in H^2_\#(\Gamma)$*, and there exists a constant* $C > 0$ *such that*

$$||u||_{H^2_\#(\Gamma)} \leq C||f||_{L^2_\#(\Gamma)}.$$

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 torus 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 $f_h \in H^1_\#(\Gamma_h)$ via the lift operator (10).

We consider u_h the weak solution of the following Poisson problem on the piecewise linear manifold Γ_h :

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

and its variational formulation, analog to (2) is

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

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

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

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 $\phi_i : \Gamma_h \to \mathbb{R}, \quad i = 1, ..., n$ such that $\phi_i(x_j) = \delta_{ij}$, (5) takes the following algebraic form

$$A_{\triangle_{\Gamma_h}} X = b_h, \tag{6}$$

where

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

$A_{\triangle_{\Gamma_h}} = (a_{ij})_{i,j=1,...,n}$, $X = {}^t(u_1, ..., u_n)$ 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 (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 Γ can be extended to a neighborhood of Γ in $\mathbb{R}^3$ using a lift operator based on the Fermi coordinates around Γ. Following [3], we define the **δ-strip around Γ** as

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

For δ small enough it is possible to define the projection $a : U_\delta \to \Gamma$ onto Γ and the distance function $d : U_\delta \to \mathbb{R}_+$ to Γ. $a(x)$ 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 Γ 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

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

where $\overrightarrow{n}(x)$ is the unit normal vector to Γ 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

$$\begin{array}{rcl} L : C(\Gamma_h) & \to & C(\Gamma) \\ u_h & \to & u_h \circ a^{-1} \end{array}, \tag{10}$$

provided

$$\Gamma_h \subset U_{\delta_{Fermi},\Gamma}. \tag{11}$$

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 Γ_h is close enough to Γ.

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

Theorem 5 (Convergence of Γ_h towards Γ). *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 trianges in Γ_h. There exists a constant c such that*

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

Once proven that Γ_h converges towards Γ, 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 trianges in Γ_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*

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

3 Numerical results for Laplace-Beltrami operator on Torus

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.

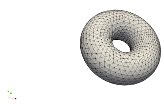
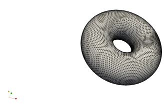
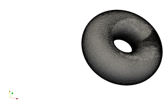
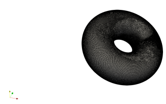
meshTorus 1	meshTorus 2	meshTorus 3	meshTorus 4
			
1022 cells	6461 cells	20006 cells	43910 cells

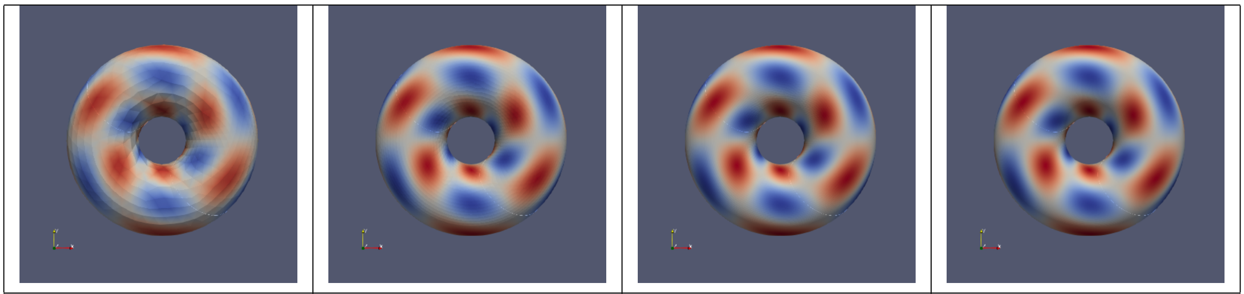
Figure 2: Mesh of domain

3.2 Visualization of the results

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

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

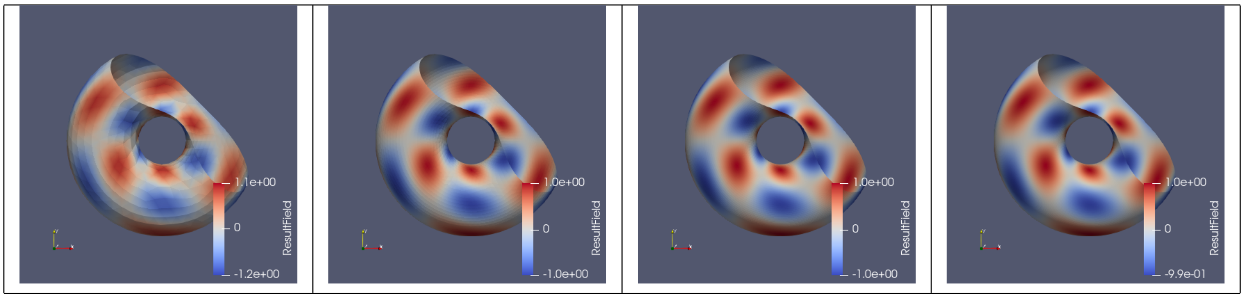
Below are visualizations of the numerical results obtained on the different meshes of picture 2.



1022 cells	6461 cells	20006 cells	43910 cells

Figure 3: Numerical results of the finite elements on the torus

Below are clipings of the previous numerical results.



1022 cells	6461 cells	20006 cells	43910 cells

Figure 4: Cliping of the numerical result on the torus

3.3 Numerical convergence of the finite element method

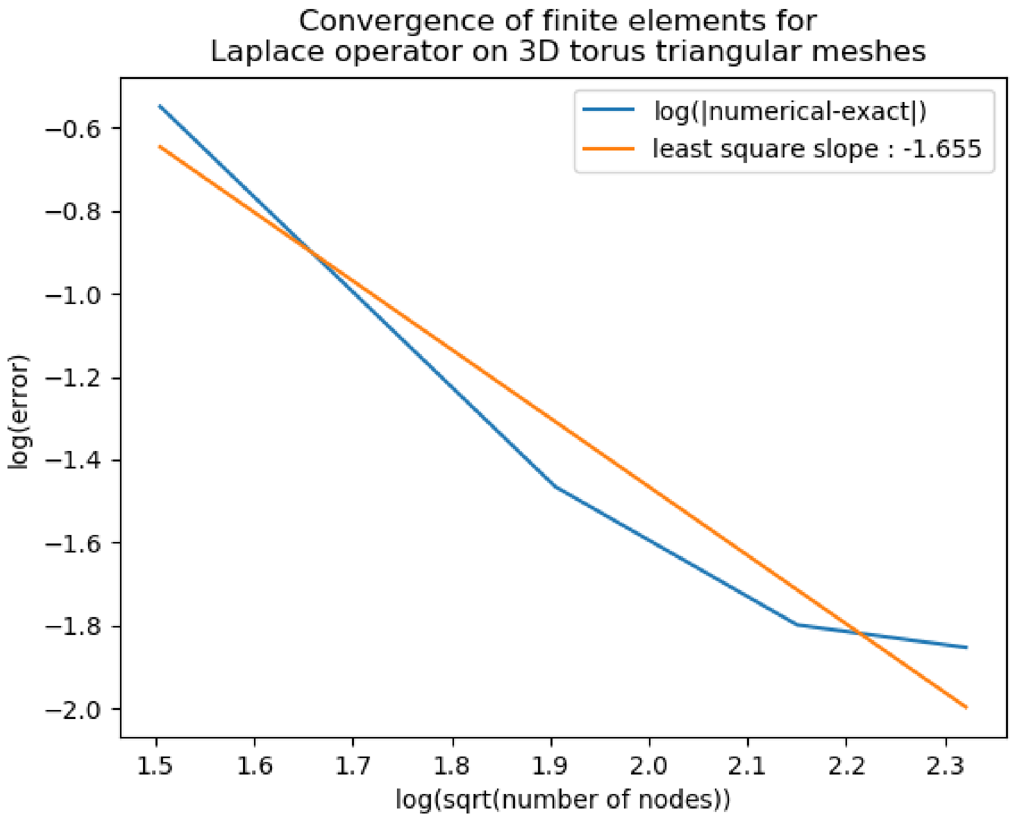


Figure 5: Convergence of the finite element method on the torus

The method converges with a numerical order of approximately 1.65.

3.4 Computational time of the finite element method

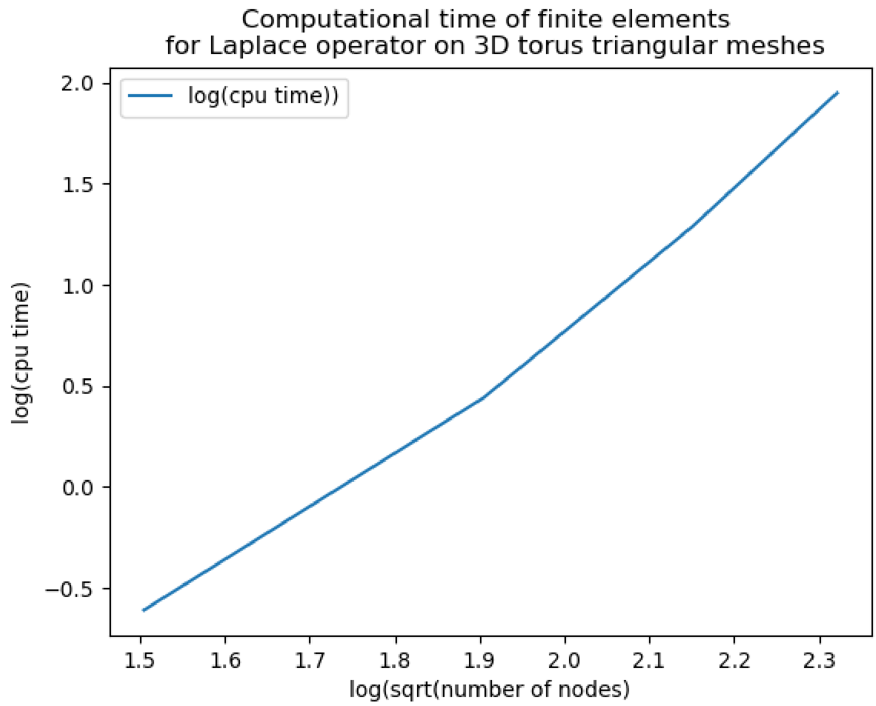


Figure 6: Computational time of the finite element method on the torus

3.5 Ploting over slice circle

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

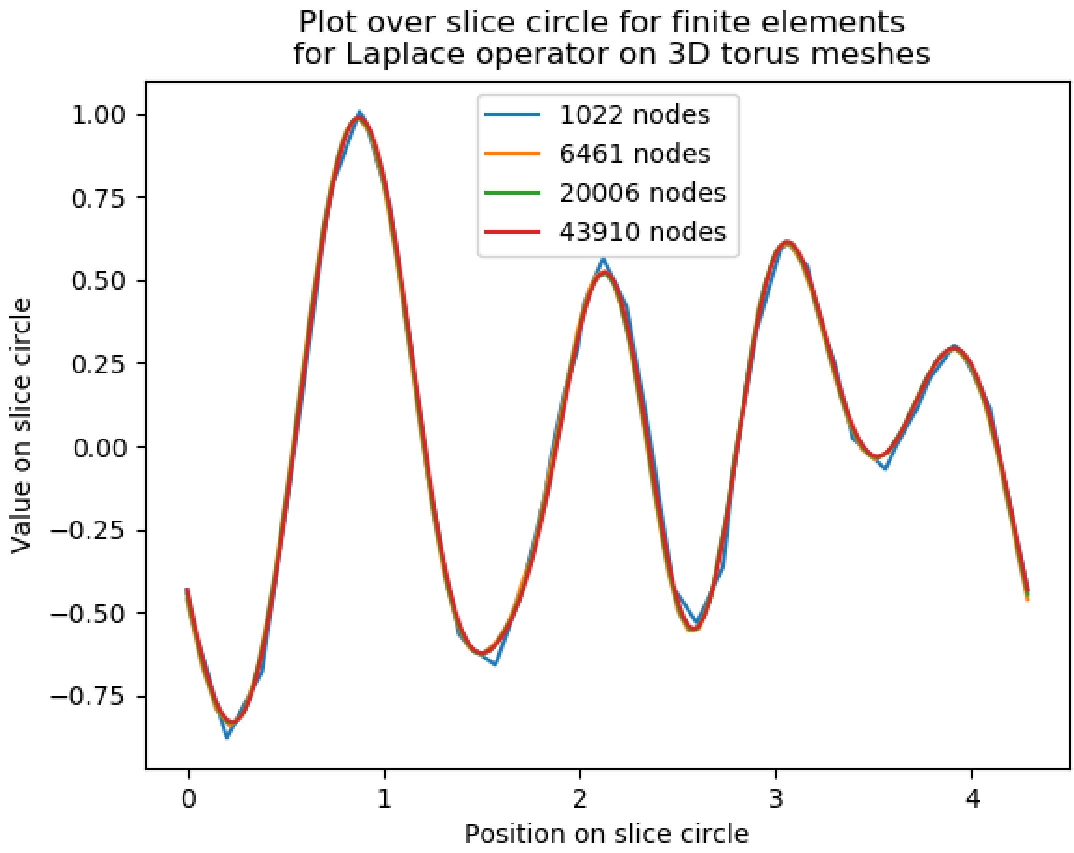


Figure 7: Convergence of the data ploted over a circle drawn on the torus

4 The script

```
# -*-coding:utf-8 -*
# ===========================================================================================
# Name          : Résolution EF de l'équation de Laplace-Beltrami -\
  triangle u = f sur un tore
# Author        : Michael Ndjinga
# Copyright     : CEA Saclay 2018
# 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
```

```
#                Référence : M. A. Olshanskii, A. Reusken, and J. Grande. A finite element method for elliptic equations
#                            on surfaces. SIAM J. Num. Anal., 47, p. 3355
#                Résolution d'un système linéaire à matrice singulière : les vecteurs constants sont dans le noyau
#================================================================================================================

import cdmath
from math import sin, cos, atan2, sqrt
import PV_routines
import VTK_routines
import paraview.simple as pvs

#Chargement du maillage triangulaire du tore
#=======================================================================================

my_mesh = cdmath.Mesh("meshTorus.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)

# Torus radii (calculation  will fail if the mesh is not correct)
R=1 #Grand rayon
r=0.6 #Petit rayon

#Discrétisation du second membre, de la solution exacte et détermination des noeuds intérieurs
#======================================================================

my_RHSfield = cdmath.Field("RHS field", cdmath.NODES, my_mesh, 1)
exactSolField = cdmath.Field("Exact solution 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

#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()

```

```
  theta=atan2(z,sqrt(x*x+y*y)-R)
  phi=atan2(y,x)

  exactSolField[i] = sin(3*phi)*cos(3*theta+ phi) # for the exact solution we use the funtion given in the article of Olshanskii, Reusken 2009, page 19
  my_RHSfield[i] = 9*sin(3*phi)*cos(3*theta+ phi)/(r*r) + (10*sin(3*phi)*cos(3*theta+ phi) + 6*cos(3*phi)*sin(3*theta+ phi))/((R+r*cos(theta))*(R+r*cos(theta))) - 3*sin(theta)*sin(3*phi)*sin(3*theta+ phi)/(r*(R+r*cos(theta))) #for the right hand side we use the function given in the article of Olshanskii, Reusken 2009, page 19
  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)
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/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()/2
  GradShapeFunc0[1]=-cellMat.partMatrix(0,1).determinant()/2
  GradShapeFunc0[2]= cellMat.partMatrix(0,2).determinant()/2
  GradShapeFunc1[0]=-cellMat.partMatrix(1,0).determinant()/2
  GradShapeFunc1[1]= cellMat.partMatrix(1,1).determinant()/2
  GradShapeFunc1[2]=-cellMat.partMatrix(1,2).determinant()/2
  GradShapeFunc2[0]= cellMat.partMatrix(2,0).determinant()/2
  GradShapeFunc2[1]=-cellMat.partMatrix(2,1).determinant()/2
  GradShapeFunc2[2]= cellMat.partMatrix(2,2).determinant()/2

  #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 système linéaire
    for k in [nodeId0,nodeId1,nodeId2] :
      Rigidite.addValue(j,k,GradShapeFuncs[j]*GradShapeFuncs[k]/Ci.getMeasure())

print("Linear system matrix building done")

# Résolution du système linéaire
#=================================
LS=cdmath.LinearSolver(Rigidite,RHS,100,1.E-6,"GMRES","ILU")#Remplacer CG par CHOLESKY pour solveur direct
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("Numerical result field", cdmath.
    NODES, my_mesh, 1)
for j in range(nbNodes):
    my_ResultField[j]=SolSyst[j];#remplissage des valeurs pour les
    noeuds intérieurs
#sauvegarde sur le disque dur du résultat dans un fichier paraview
my_ResultField.writeVTK("FiniteElementsOnTorusPoisson")

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

#Calcul de l'erreur commise par rapport à la solution exacte
#===========================================================
max_sol_exacte=exactSolField.getNormEuclidean().max()
erreur_max=(exactSolField - my_ResultField).getNormEuclidean().max
    ()
max_sol_num=my_ResultField.max()
min_sol_num=my_ResultField.min()

print("Relative error =  max(| exact solution - numerical solution
    |)/max(| exact solution |) = ",erreur_max/max_sol_exacte)
print("Maximum numerical solution = ", max_sol_num, " Minimum
    numerical solution = ", min_sol_num)
print("Maximum exact solution = ", exactSolField.max(), " Minimum
    exact solution = ", exactSolField.min())

#Postprocessing :
#================
# Save 3D picture
PV_routines.Save_PV_data_to_picture_file("
    FiniteElementsOnTorusPoisson"+'_0.vtu',"Numerical result field"
    ,'NODES',"FiniteElementsOnTorusPoisson")
resolution=100
VTK_routines.Clip_VTK_data_to_VTK("FiniteElementsOnTorusPoisson"+'
    _0.vtu',"Clip_VTK_data_to_VTK_"+ "FiniteElementsOnTorusPoisson"
    +'_0.vtu',[0.25,0.25,0.25], [-0.5,-0.5,-0.5],resolution )
PV_routines.Save_PV_data_to_picture_file("Clip_VTK_data_to_VTK_"+"
    FiniteElementsOnTorusPoisson"+'_0.vtu',"Numerical result field"
    ,'NODES',"Clip_VTK_data_to_VTK_"+"FiniteElementsOnTorusPoisson"
    )

# Plot  over slice circle
finiteElementsOnTorus_0vtu = pvs.XMLUnstructuredGridReader(FileName
    =["FiniteElementsOnTorusPoisson"+'_0.vtu'])
slice1 = pvs.Slice(Input=finiteElementsOnTorus_0vtu)
slice1.SliceType.Normal = [0.5, 0.5, 0.5]
renderView1 = pvs.GetActiveViewOrCreate('RenderView')
finiteElementsOnTorus_0vtuDisplay = pvs.Show(
    finiteElementsOnTorus_0vtu, renderView1)
pvs.ColorBy(finiteElementsOnTorus_0vtuDisplay, ('POINTS', '
    Numerical result field'))
slice1Display = pvs.Show(slice1, renderView1)
pvs.SaveScreenshot("./FiniteElementsOnTorusPoisson"+"_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 = ['Numerical result field (1)']
pvs.SaveScreenshot("./FiniteElementsOnTorusPoisson"+"_PlotOnSortedLine_"+'.png', magnification=1, quality=100, view=lineChartView2)
pvs.Delete(lineChartView2)

assert erreur_max/max_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 Riemmannian 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