

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

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

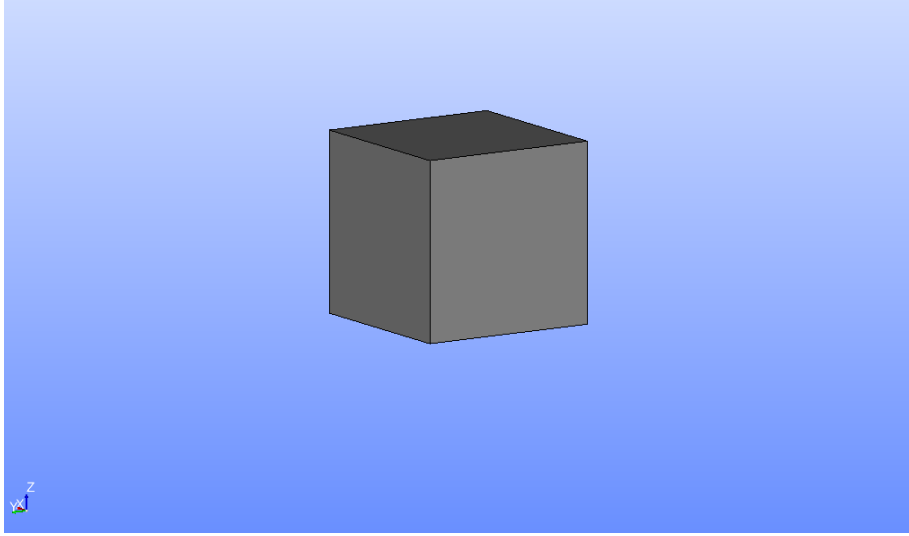


Figure 1: The unit cube in SALOME CAO module

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 $\Delta_\Gamma = \nabla \cdot \vec{\nabla}$, 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

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

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

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

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

$$-\Delta_{\Gamma} u = f, \tag{1}$$

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

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 Poisson 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:

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

2.1.2 Existence of the 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** for Lipschitz manifolds proved in [9].

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. 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

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

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.

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

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

$$A_{\Delta_{\Gamma_h}} X = b_h, \quad (4)$$

where

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

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

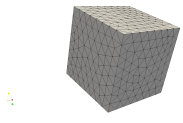
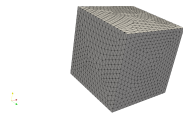
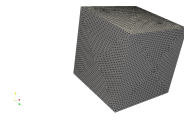
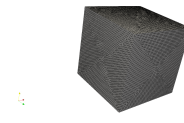
meshCubeSkin 1	meshCubeSkin 2	meshCubeSkin 3	meshCubeSkin 4
			
412 cells	1923 cells	7042 cells	13225 cells

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

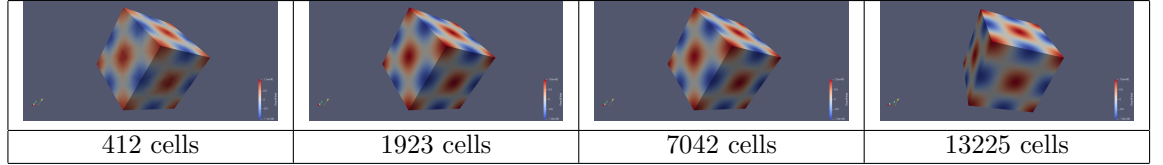


Figure 3: Numerical results of the finite elements on the unit cube skin

Below are clipings of the previous numerical results.

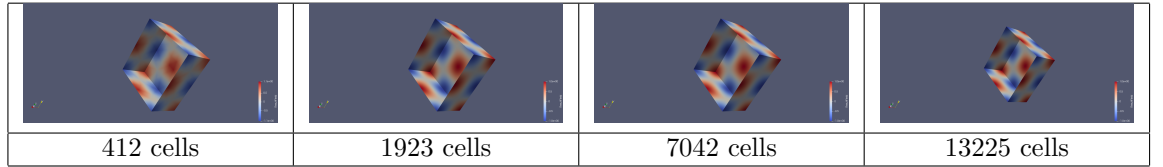


Figure 4: Clipping of the numerical result on the unit cube skin

3.3 Numerical convergence of the finite element method

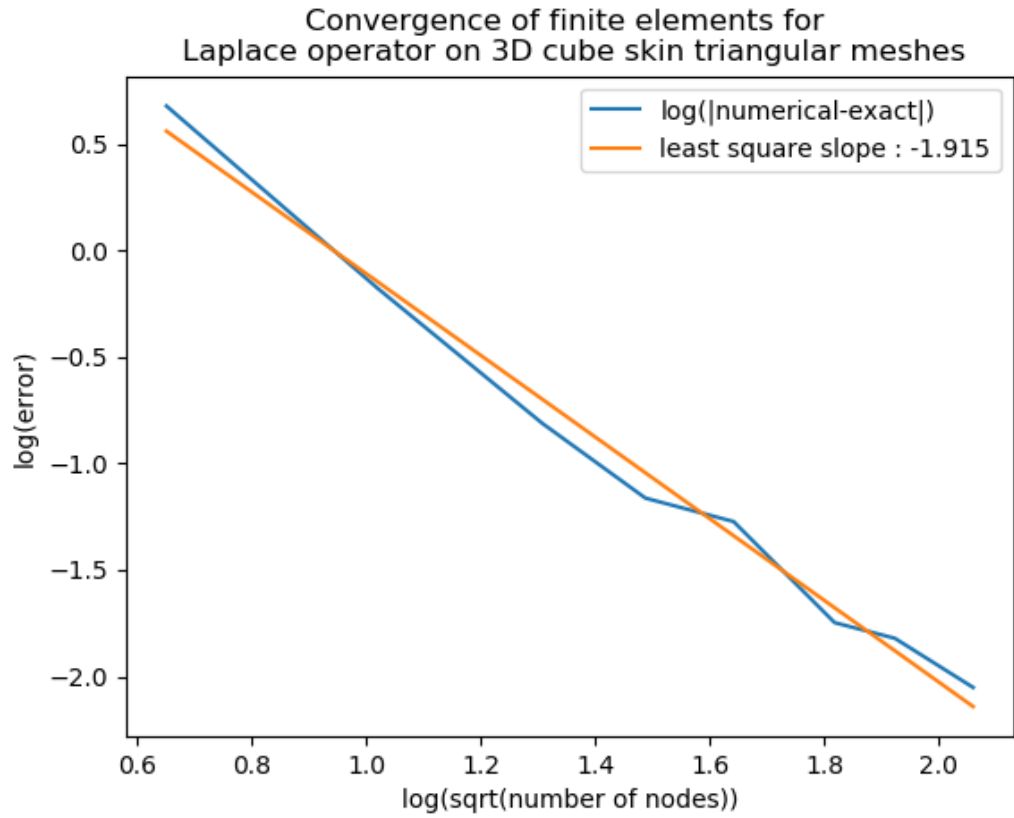


Figure 5: Convergence of the finite element method on the cube skin

The method converges with a numerical order of approximately 1.91.

3.4 Computational time of the finite element method

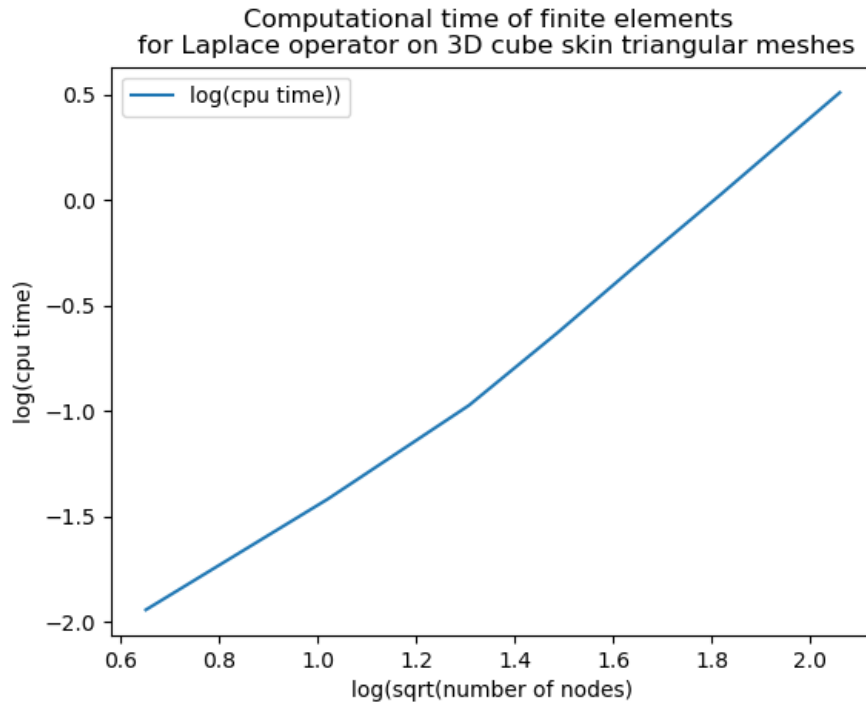


Figure 6: Computational time of the finite element method on the cube skin

4 The script

```

1 # -*-coding:utf-8 -*-
2 #
3 # Name      : Résolution EF de l'équation de Laplace-Beltrami -\
4 #           triangle u = f sur la frontière d'un cube
5 # Author    : Michael Ndjinga
6 # Copyright  : CEA Saclay 2021
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 #            Création et sauvegarde du champ résultant ainsi que
11 #            du champ second membre en utilisant la librairie CDMATH
12 #            Résolution d'un système linéaire à matrice singuliè
13 #            re : les vecteurs constants sont dans le noyau
14 #            Comparaison de la solution numérique avec la
15 #            solution exacte définie face par face : u(x,y,z)= cos(2*pi*x)*
16 #            cos(2*pi*y)*cos(2*pi*z)
17 #
18 #
19 #
20 #
21 #
22 #
23 #
24 #
25 #
26 #
27 #
28 #
29 #
30 #
31 #
32 #
33 #
34 #
35 #
36 #
37 #
38 #
39 #
40 #
41 #
42 #
43 #
44 #
45 #
46 #
47 #
48 #
49 #
50 #
51 #
52 #
53 #
54 #
55 #
56 #
57 #
58 #
59 #
60 #
61 #
62 #
63 #
64 #
65 #
66 #
67 #
68 #
69 #
70 #
71 #
72 #
73 #
74 #
75 #
76 #
77 #
78 #
79 #
80 #
81 #
82 #
83 #
84 #
85 #
86 #
87 #
88 #
89 #
90 #
91 #
92 #
93 #
94 #
95 #
96 #
97 #
98 #
99 #
100 #
101 #
102 #
103 #
104 #
105 #
106 #
107 #
108 #
109 #
110 #
111 #
112 #
113 #
114 #
115 #
116 #
117 #
118 #
119 #
120 #
121 #
122 #
123 #
124 #
125 #
126 #
127 #
128 #
129 #
130 #
131 #
132 #
133 #
134 #
135 #
136 #
137 #
138 #
139 #
140 #
141 #
142 #
143 #
144 #
145 #
146 #
147 #
148 #
149 #
150 #
151 #
152 #
153 #
154 #
155 #
156 #
157 #
158 #
159 #
160 #
161 #
162 #
163 #
164 #
165 #
166 #
167 #
168 #
169 #
170 #
171 #
172 #
173 #
174 #
175 #
176 #
177 #
178 #
179 #
180 #
181 #
182 #
183 #
184 #
185 #
186 #
187 #
188 #
189 #
190 #
191 #
192 #
193 #
194 #
195 #
196 #
197 #
198 #
199 #
200 #
201 #
202 #
203 #
204 #
205 #
206 #
207 #
208 #
209 #
210 #
211 #
212 #
213 #
214 #
215 #
216 #
217 #
218 #
219 #
220 #
221 #
222 #
223 #
224 #
225 #
226 #
227 #
228 #
229 #
230 #
231 #
232 #
233 #
234 #
235 #
236 #
237 #
238 #
239 #
240 #
241 #
242 #
243 #
244 #
245 #
246 #
247 #
248 #
249 #
250 #
251 #
252 #
253 #
254 #
255 #
256 #
257 #
258 #
259 #
260 #
261 #
262 #
263 #
264 #
265 #
266 #
267 #
268 #
269 #
270 #
271 #
272 #
273 #
274 #
275 #
276 #
277 #
278 #
279 #
280 #
281 #
282 #
283 #
284 #
285 #
286 #
287 #
288 #
289 #
290 #
291 #
292 #
293 #
294 #
295 #
296 #
297 #
298 #
299 #
300 #
301 #
302 #
303 #
304 #
305 #
306 #
307 #
308 #
309 #
310 #
311 #
312 #
313 #
314 #
315 #
316 #
317 #
318 #
319 #
320 #
321 #
322 #
323 #
324 #
325 #
326 #
327 #
328 #
329 #
330 #
331 #
332 #
333 #
334 #
335 #
336 #
337 #
338 #
339 #
340 #
341 #
342 #
343 #
344 #
345 #
346 #
347 #
348 #
349 #
350 #
351 #
352 #
353 #
354 #
355 #
356 #
357 #
358 #
359 #
360 #
361 #
362 #
363 #
364 #
365 #
366 #
367 #
368 #
369 #
370 #
371 #
372 #
373 #
374 #
375 #
376 #
377 #
378 #
379 #
380 #
381 #
382 #
383 #
384 #
385 #
386 #
387 #
388 #
389 #
390 #
391 #
392 #
393 #
394 #
395 #
396 #
397 #
398 #
399 #
400 #
401 #
402 #
403 #
404 #
405 #
406 #
407 #
408 #
409 #
410 #
411 #
412 #
413 #
414 #
415 #
416 #
417 #
418 #
419 #
420 #
421 #
422 #
423 #
424 #
425 #
426 #
427 #
428 #
429 #
430 #
431 #
432 #
433 #
434 #
435 #
436 #
437 #
438 #
439 #
440 #
441 #
442 #
443 #
444 #
445 #
446 #
447 #
448 #
449 #
450 #
451 #
452 #
453 #
454 #
455 #
456 #
457 #
458 #
459 #
460 #
461 #
462 #
463 #
464 #
465 #
466 #
467 #
468 #
469 #
470 #
471 #
472 #
473 #
474 #
475 #
476 #
477 #
478 #
479 #
480 #
481 #
482 #
483 #
484 #
485 #
486 #
487 #
488 #
489 #
490 #
491 #
492 #
493 #
494 #
495 #
496 #
497 #
498 #
499 #
500 #
501 #
502 #
503 #
504 #
505 #
506 #
507 #
508 #
509 #
510 #
511 #
512 #
513 #
514 #
515 #
516 #
517 #
518 #
519 #
520 #
521 #
522 #
523 #
524 #
525 #
526 #
527 #
528 #
529 #
530 #
531 #
532 #
533 #
534 #
535 #
536 #
537 #
538 #
539 #
540 #
541 #
542 #
543 #
544 #
545 #
546 #
547 #
548 #
549 #
550 #
551 #
552 #
553 #
554 #
555 #
556 #
557 #
558 #
559 #
560 #
561 #
562 #
563 #
564 #
565 #
566 #
567 #
568 #
569 #
570 #
571 #
572 #
573 #
574 #
575 #
576 #
577 #
578 #
579 #
580 #
581 #
582 #
583 #
584 #
585 #
586 #
587 #
588 #
589 #
590 #
591 #
592 #
593 #
594 #
595 #
596 #
597 #
598 #
599 #
600 #
601 #
602 #
603 #
604 #
605 #
606 #
607 #
608 #
609 #
610 #
611 #
612 #
613 #
614 #
615 #
616 #
617 #
618 #
619 #
620 #
621 #
622 #
623 #
624 #
625 #
626 #
627 #
628 #
629 #
630 #
631 #
632 #
633 #
634 #
635 #
636 #
637 #
638 #
639 #
640 #
641 #
642 #
643 #
644 #
645 #
646 #
647 #
648 #
649 #
650 #
651 #
652 #
653 #
654 #
655 #
656 #
657 #
658 #
659 #
660 #
661 #
662 #
663 #
664 #
665 #
666 #
667 #
668 #
669 #
670 #
671 #
672 #
673 #
674 #
675 #
676 #
677 #
678 #
679 #
680 #
681 #
682 #
683 #
684 #
685 #
686 #
687 #
688 #
689 #
690 #
691 #
692 #
693 #
694 #
695 #
696 #
697 #
698 #
699 #
700 #
701 #
702 #
703 #
704 #
705 #
706 #
707 #
708 #
709 #
710 #
711 #
712 #
713 #
714 #
715 #
716 #
717 #
718 #
719 #
720 #
721 #
722 #
723 #
724 #
725 #
726 #
727 #
728 #
729 #
730 #
731 #
732 #
733 #
734 #
735 #
736 #
737 #
738 #
739 #
740 #
741 #
742 #
743 #
744 #
745 #
746 #
747 #
748 #
749 #
750 #
751 #
752 #
753 #
754 #
755 #
756 #
757 #
758 #
759 #
760 #
761 #
762 #
763 #
764 #
765 #
766 #
767 #
768 #
769 #
770 #
771 #
772 #
773 #
774 #
775 #
776 #
777 #
778 #
779 #
780 #
781 #
782 #
783 #
784 #
785 #
786 #
787 #
788 #
789 #
790 #
791 #
792 #
793 #
794 #
795 #
796 #
797 #
798 #
799 #
800 #
801 #
802 #
803 #
804 #
805 #
806 #
807 #
808 #
809 #
810 #
811 #
812 #
813 #
814 #
815 #
816 #
817 #
818 #
819 #
820 #
821 #
822 #
823 #
824 #
825 #
826 #
827 #
828 #
829 #
830 #
831 #
832 #
833 #
834 #
835 #
836 #
837 #
838 #
839 #
840 #
841 #
842 #
843 #
844 #
845 #
846 #
847 #
848 #
849 #
850 #
851 #
852 #
853 #
854 #
855 #
856 #
857 #
858 #
859 #
860 #
861 #
862 #
863 #
864 #
865 #
866 #
867 #
868 #
869 #
870 #
871 #
872 #
873 #
874 #
875 #
876 #
877 #
878 #
879 #
880 #
881 #
882 #
883 #
884 #
885 #
886 #
887 #
888 #
889 #
890 #
891 #
892 #
893 #
894 #
895 #
896 #
897 #
898 #
899 #
900 #
901 #
902 #
903 #
904 #
905 #
906 #
907 #
908 #
909 #
910 #
911 #
912 #
913 #
914 #
915 #
916 #
917 #
918 #
919 #
920 #
921 #
922 #
923 #
924 #
925 #
926 #
927 #
928 #
929 #
930 #
931 #
932 #
933 #
934 #
935 #
936 #
937 #
938 #
939 #
940 #
941 #
942 #
943 #
944 #
945 #
946 #
947 #
948 #
949 #
950 #
951 #
952 #
953 #
954 #
955 #
956 #
957 #
958 #
959 #
960 #
961 #
962 #
963 #
964 #
965 #
966 #
967 #
968 #
969 #
970 #
971 #
972 #
973 #
974 #
975 #
976 #
977 #
978 #
979 #
980 #
981 #
982 #
983 #
984 #
985 #
986 #
987 #
988 #
989 #
990 #
991 #
992 #
993 #
994 #
995 #
996 #
997 #
998 #
999 #
1000 #

```

```

12 import cdmath
13 from math import cos, pi
14 import numpy as np
15 import PV_routines
16 import VTK_routines
17 import paraview.simple as pvs
18
19 #Chargement du maillage triangulaire de la frontière du cube unité
    [0,1]x[0,1]x[0,1]
20 #
    =====
21 my_mesh = cdmath.Mesh("meshCubeSkin.med")
22 if(not my_mesh.isTriangular()) :
23     raise ValueError("Wrong cell types : mesh is not made of
        triangles")
24 if(my_mesh.getMeshDimension()!=2) :
25     raise ValueError("Wrong mesh dimension : expected a surface of
        dimension 2")
26 if(my_mesh.getSpaceDimension()!=3) :
27     raise ValueError("Wrong space dimension : expected a space of
        dimension 3")
28
29 nbNodes = my_mesh.getNumberOfNodes()
30 nbCells = my_mesh.getNumberOfCells()
31
32 print("Mesh building/loading done")
33 print("nb of nodes=", nbNodes)
34 print("nb of cells=", nbCells)
35
36 #Discrétisation du second membre et détermination des noeuds inté
    rieurs
37 #
    =====
38 my_RHSfield = cdmath.Field("RHS field", cdmath.NODES, my_mesh, 1)
39 maxNbNeighbours = 0#This is to determine the number of non zero
    coefficients in the sparse finite element rigidity matrix
40
41 eps=1e-6
42 #parcours des noeuds pour discrétisation du second membre et
    extraction du nb max voisins d'un noeud
43 for i in range(nbNodes):
44     Ni=my_mesh.getNode(i)
45     x = Ni.x()
46     y = Ni.y()
47     z = Ni.z()
48
49     my_RHSfield[i]= 8*pi*pi*cos(2*pi*x)*cos(2*pi*y)*cos(2*pi*z)
50
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)
        #true only for planar cells, otherwise use function Ni.
        getNumberOfEdges()
55
56 print("Right hand side discretisation done")
57 print("Max nb of neighbours=", maxNbNeighbours)
58 print("Integral of the RHS", my_RHSfield.integral(0))
59
60 # Construction de la matrice de rigidité et du vecteur second

```



```

        membre du système linéaire
61 #
        =====

62 Rigidite=cdmath.SparseMatrixPetsc(nbNodes,nbNodes,maxNbNeighbours)#
        warning : third argument is number of non zero coefficients
        per line
63 RHS=cdmath.Vector(nbNodes)
64
65 # Vecteurs gradient de la fonction de forme associée à chaque noeud
        d'un triangle
66 GradShapeFunc0=cdmath.Vector(3)
67 GradShapeFunc1=cdmath.Vector(3)
68 GradShapeFunc2=cdmath.Vector(3)
69
70 normalFace0=cdmath.Vector(3)
71 normalFace1=cdmath.Vector(3)
72
73 #On parcourt les triangles du domaine
74 for i in range(nbCells):
75
76     Ci=my_mesh.getCell(i)
77
78     #Contribution à la matrice de rigidité
79     nodeId0=Ci.getNodeId(0)
80     nodeId1=Ci.getNodeId(1)
81     nodeId2=Ci.getNodeId(2)
82     N0=my_mesh.getNode(nodeId0)
83     N1=my_mesh.getNode(nodeId1)
84     N2=my_mesh.getNode(nodeId2)
85
86     #Build normal to cell Ci
87     normalFace0[0]=Ci.getNormalVector(0,0)
88     normalFace0[1]=Ci.getNormalVector(0,1)
89     normalFace0[2]=Ci.getNormalVector(0,2)
90     normalFace1[0]=Ci.getNormalVector(1,0)
91     normalFace1[1]=Ci.getNormalVector(1,1)
92     normalFace1[2]=Ci.getNormalVector(1,2)
93
94     normalCell = normalFace0.crossProduct(normalFace1)
95     normalCell = normalCell*(1/normalCell.norm())
96
97     cellMat=cdmath.Matrix(4)
98     cellMat[0,0]=N0.x()
99     cellMat[0,1]=N0.y()
100    cellMat[0,2]=N0.z()
101    cellMat[1,0]=N1.x()
102    cellMat[1,1]=N1.y()
103    cellMat[1,2]=N1.z()
104    cellMat[2,0]=N2.x()
105    cellMat[2,1]=N2.y()
106    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()*0.5

```

```

117 GradShapeFunc0[1]=-cellMat.partMatrix(0,1).determinant()*0.5
118 GradShapeFunc0[2]= cellMat.partMatrix(0,2).determinant()*0.5
119 GradShapeFunc1[0]=-cellMat.partMatrix(1,0).determinant()*0.5
120 GradShapeFunc1[1]= cellMat.partMatrix(1,1).determinant()*0.5
121 GradShapeFunc1[2]=-cellMat.partMatrix(1,2).determinant()*0.5
122 GradShapeFunc2[0]= cellMat.partMatrix(2,0).determinant()*0.5
123 GradShapeFunc2[1]=-cellMat.partMatrix(2,1).determinant()*0.5
124 GradShapeFunc2[2]= cellMat.partMatrix(2,2).determinant()*0.5
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 # Conditionnement de la matrice de rigidité
142 #=====
143 cond = Rigidite.getConditionNumber(True)
144 print("Condition number is ",cond)
145
146 # Résolution du système linéaire
147 #=====
148 LS=cdmath.LinearSolver(Rigidite,RHS,100,1.E-6,"GMRES","ILU")
149 LS.setMatrixIsSingular()#En raison de l'absence de bord
150 SolSyst=LS.solve()
151 print("Preconditioner used : ", LS.getNameOfPc() )
152 print("Number of iterations used : ", LS.getNumberOfIter() )
153 print("Final residual : ", LS.getResidu() )
154 print("Linear system solved")
155
156 # Création du champ résultat
157 #=====
158 my_ResultField = cdmath.Field("ResultField", cdmath.NODES, my_mesh,
    1)
159 for j in range(nbNodes):
160     my_ResultField[j]=SolSyst[j];#remplissage des valeurs issues du
    système linéaire dans le champs résultat
161 #sauvegarde sur le disque dur du résultat dans un fichier paraview
162 my_ResultField.writeVTK("FiniteElementsOnCubeSkinPoisson")
163 my_RHSfield.writeVTK("RHS_CubeSkinPoisson")
164
165 print("Integral of the numerical solution", my_ResultField.integral
    (0))
166 print("Numerical solution of Poisson equation on a cube skin using
    finite elements done")
167
168 #Calcul de l'erreur commise par rapport à la solution exacte
169 #=====

```

```

170 #The following formulas use the fact that the exact solution is
    equal the right hand side divided by 8*pi*pi
171 max_abs_sol_exacte=0
172 erreur_abs=0
173 max_sol_num=0
174 min_sol_num=0
175 for i in range(nbNodes) :
176     if max_abs_sol_exacte < abs(my_RHSfield[i]) :
177         max_abs_sol_exacte = abs(my_RHSfield[i])
178     if erreur_abs < abs(my_RHSfield[i]/(8*pi*pi) - my_ResultField[
        i]) :
179         erreur_abs = abs(my_RHSfield[i]/(8*pi*pi) - my_ResultField[
            i])
180     if max_sol_num < my_ResultField[i] :
181         max_sol_num = my_ResultField[i]
182     if min_sol_num > my_ResultField[i] :
183         min_sol_num = my_ResultField[i]
184 max_abs_sol_exacte = max_abs_sol_exacte/(8*pi*pi)
185
186 print("Relative error = max(| exact solution - numerical solution
    |)/max(| exact solution |) = ",erreur_abs/max_abs_sol_exacte)
187 print("Maximum numerical solution = ", max_sol_num, " Minimum
    numerical solution = ", min_sol_num)
188 print("Maximum exact solution = ", my_RHSfield.max()/(8*pi*pi), "
    Minimum exact solution = ", my_RHSfield.min()/(8*pi*pi) )
189
190 #Postprocessing :
191 #=====
192 # save 3D picture
193 PV_routines.Save_PV_data_to_picture_file("
    FiniteElementsOnCubeSkinPoisson"+'_0.vtu',"ResultField",'NODES'
    ,"FiniteElementsOnCubeSkinPoisson")
194 resolution=100
195 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 )
196 PV_routines.Save_PV_data_to_picture_file("Clip_VTK_data_to_VTK_"+"
    FiniteElementsOnCubeSkinPoisson"+'_0.vtu',"ResultField",'NODES'
    ,"Clip_VTK_data_to_VTK_"+"FiniteElementsOnCubeSkinPoisson")
197
198 # Plot over slice circle
199 finiteElementsOnCubeSkin_0vtu = pvs.XMLUnstructuredGridReader(
    FileName=["FiniteElementsOnCubeSkinPoisson"+'_0.vtu'])
200 slice1 = pvs.Slice(Input=finiteElementsOnCubeSkin_0vtu)
201 slice1.SliceType.Normal = [0, 1, 0]
202 renderView1 = pvs.GetActiveViewOrCreate('RenderView')
203 finiteElementsOnCubeSkin_0vtuDisplay = pvs.Show(
    finiteElementsOnCubeSkin_0vtu, renderView1)
204 pvs.ColorBy(finiteElementsOnCubeSkin_0vtuDisplay, ('POINTS', '
    ResultField'))
205 slice1Display = pvs.Show(slice1, renderView1)
206 pvs.SaveScreenshot("./FiniteElementsOnCubeSkinPoisson"+ "_Slice"+'.
    png', magnification=1, quality=100, view=renderView1)
207 plotOnSortedLines1 = pvs.PlotOnSortedLines(Input=slice1)
208 lineChartView2 = pvs.CreateView('XYChartView')
209 plotOnSortedLines1Display = pvs.Show(plotOnSortedLines1,
    lineChartView2)
210 plotOnSortedLines1Display.UseIndexForXAxis = 0
211 plotOnSortedLines1Display.XArrayName = 'arc_length'
212 plotOnSortedLines1Display.SeriesVisibility = ['ResultField (1)']
213 pvs.SaveScreenshot("./FiniteElementsOnCubeSkinPoisson"+"

```

```

214     _PlotOnSortedLine_ "+"'.png', magnification=1, quality=100, view=
        lineChartView2)
215 pvs.Delete(lineChartView2)
216 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 Academiæ Scientiarum Fennicæ, 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>