Two dimensional Poisson and Helmholtz equations on Dirichlet boundaries conditions

Principles of Programming with Python

Assessment Report

Candidate Number: 1051811

July 2021

Contents

1 Preface

1.1 What's the assessment?

The goal of the assessment report is to guide the reader through a Python package that solves the 2D Poisson equation and 2D Helmholtz equation on a square domain with Dirichlet

boundaries conditions using finite element method with piecewise linear functions.
The reports aims to describe the class of PDE, the discretisation employed to solve them and the Python code written.

2 Introduction to the mathematical problem

We present the PDE problems that the package solves. The first PDE is the Poisson equation with homogeneous Dirichlet boundaries conditions. The mathematical problem is to find $u(x, y)$ such that for an arbitrary scalar function $f(x, y)$ and an arbitrary square domain $\Omega \subset \mathbb{R}^2$ once has:

$$\begin{cases} -\Delta u = f & \text{in} \quad \Omega \\ u = 0 & \text{on} \quad \partial\Omega \end{cases} \tag{1}$$

Δ is the Laplacian operator, namely for the scalar function $u(x, y)$ we have

$$-\Delta u = -\nabla \cdot \nabla u = -\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2}$$

The domain Ω is an arbitrary square domain; more precisely for arbitrary real constants a, b, c, d such that $b - a = d - c$ we assume

$$\Omega = \{(x, y) \in \mathbb{R}^2 \quad | \quad a \leq x \leq b \quad \wedge \quad c \leq y \leq d \quad \}$$

In other words Ω is a square with arbitrary vertices meanwhile we denoted $\partial\Omega$ as the boundary of the square. The problem is one of the most straight forward case where a solution can be approximated using finite element method.

2.1 Mathematical formulation for Poisson

The mathematical formulation on this section follows Lecture Notes of Finite Element Methods for Partial Differential Equations by Endre Suli [1] and Finite Element Method for PDEs by Patrick Farrell [2]. Assume the following definitions:

$$H_0^1(\Omega) = \{v \in H^1(\Omega) \quad | \quad v = 0 \quad on \quad \partial\Omega\}$$

$$H^1(\Omega) = \{v \in L^2(\Omega) \quad | \quad \nabla v \in L^2(\Omega)\}$$

$$L^2(\Omega) = \{v : \Omega \to \mathbb{R} \quad | \quad \|v\|_{L^2(\Omega)} < \infty\}$$

$$\|v\|_{L^2(\Omega)} = \Big(\int_\Omega |v|^2\Big)^{1/2}$$

We now summarize the mathematical idea and not go into details of functional analysis or finite element method because it is a very well known problem and goes beyond the aim of this assessment.
The weak formulation of the Poisson equation with Dirichlet boundary conditions on a closed domain $\Omega \subset \mathbb{R}^2$ consists in

$$\text{Find } u \in H_0^1(\Omega) \text{ such that } \forall v \in H_0^1(\Omega) \text{ once has: } \int_\Omega \nabla u \cdot \nabla v = \int_\Omega f v \tag{2}$$

By introducing the bi-linear form

$$a(u, v) := \int_\Omega \nabla u \cdot \nabla v = \int_\Omega \left(\frac{\partial u}{\partial x}\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\frac{\partial v}{\partial y}\right)$$

and the linear form

$$l(v) := \int_\Omega fv$$

we recast to the problem

$$\text{Find } u \in H_0^1(\Omega) \text{ such that } \forall v \in H_0^1(\Omega) \text{ once has: } a(u,v) = l(v) \tag{3}$$

If we now restrict the infinite function space $H_0^1(\Omega)$ to a finite function subspace V_h we can say that there exists functions $\phi_1(x,y), \phi_2(x,y), ..., \phi_n(x,y) \in H_0^1(\Omega)$ such that

$$V_h = \text{span}\{\phi_1, \phi_2, ..., \phi_n\}$$

For our purpose we consider linear piece-wise functions. The problem becomes

$$\text{Find } u \in H_0^1(\Omega) \text{ such that } \forall j = 1, ..., n \text{ once has: } a(u,\phi_j) = l(\phi_j) \tag{4}$$

Before going into more details we need to divide the square domain Ω into an arbitrary number of cells. We triangulate the domain, as it is mentioned at page 30 of [1], with equilateral triangles of diameter h; see Figure 1. We associate a linear piece-wise function

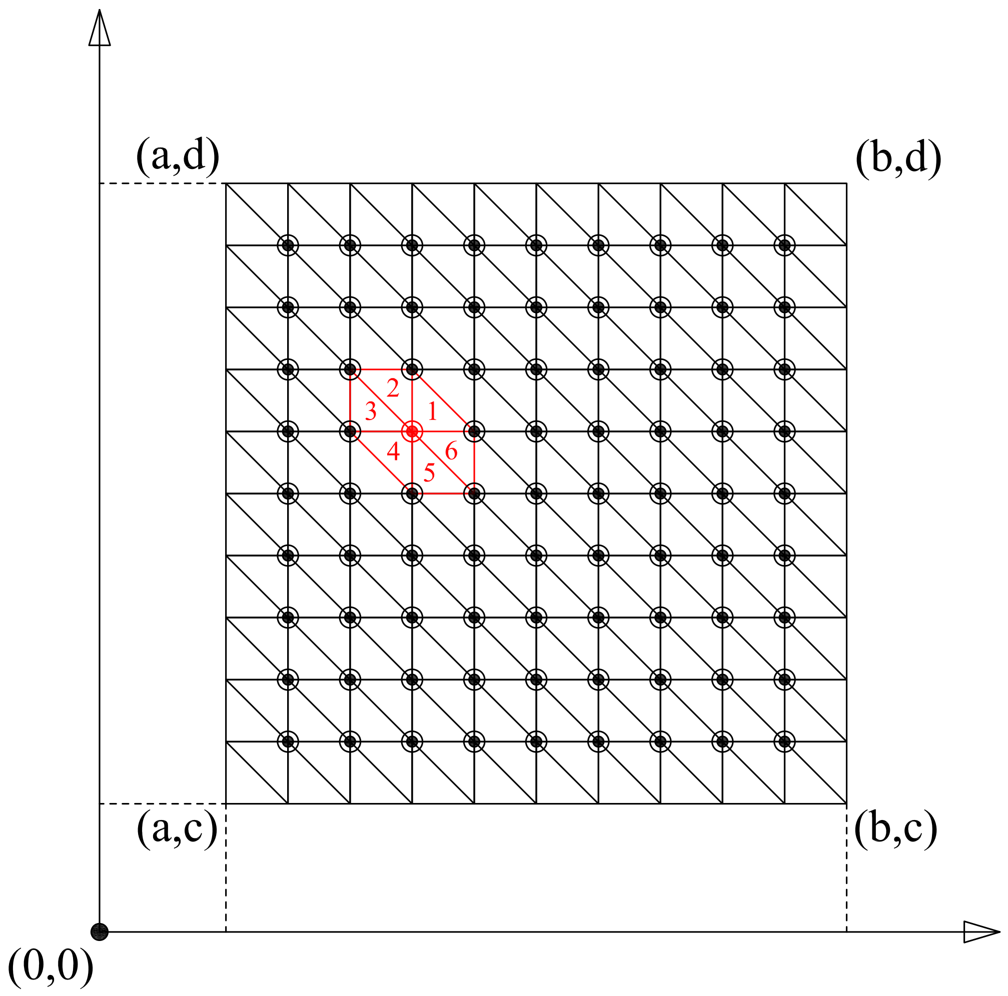


Figure 1: Triangulation of Ω. The highlighted red part corresponds to the support of an arbitrary linear piece-wise function $\phi_j(x,y)$ associated to the j-th interior node (also in red) of the triangulation.

$\phi_i(x,y)$ to each interior node i of the triangulation. Assume that the interior nodes are enumerated from left-bottom to right-top. It becomes clear that the dimension of V_h is equal to the number of interior nodes we pick. Similarly the diameter length h of each triangle decreases as we increase the dimension of V_h. Each interior node i has Cartesian coordinate $(x_i, y_i) \in \mathbb{R}^2$ and the linear piece-wise basis function is

$$\phi_i(x,y) = \begin{cases} 1 - \frac{x-x_i}{h} - \frac{y-y_i}{h} & \text{if} \quad (x,y) \in \mathbf{1} \\ 1 - \frac{y-y_i}{h} & \text{if} \quad (x,y) \in \mathbf{2} \\ 1 + \frac{x-x_i}{h} & \text{if} \quad (x,y) \in \mathbf{3} \\ 1 + \frac{x-x_i}{h} + \frac{y-y_i}{h} & \text{if} \quad (x,y) \in \mathbf{4} \\ 1 + \frac{y-y_i}{h} & \text{if} \quad (x,y) \in \mathbf{5} \\ 1 - \frac{x-x_i}{h} & \text{if} \quad (x,y) \in \mathbf{6} \\ 0 & \text{else} \end{cases} \tag{5}$$

Notice how this function maps its associated node to 1 and all the other node to 0, furthermore $\mathbf{1}, \mathbf{2}, ..., \mathbf{6}$ are represented in Figure 1 and their union forms the support of ϕ_i.
Instead of finding u we find an approximation $u_h \in V_h$. If we do so by recalling $V_h = \text{span}\{\phi_1, \phi_2, ..., \phi_{N(h)}\}$ we notice there exist real constants $U_1, ..., U_{N(h)}$ such that

$$u_h(x,y) = U_1\phi_1 + U_2\phi_2 + ... + U_{N(h)}\phi_{N(h)} = \sum_{i=1}^{N(h)} U_i\phi_i(x,y)$$

Mathematical approximated problem becomes:

$$\text{Find } U_1, ..., U_{N(h)} \text{ such that } \forall j = 1, .., N(h) \text{ once has: } \sum_{i=1}^{N(h)} U_i a(\phi_i, \phi_j) = l(\phi_j) \tag{6}$$

Which in vector form becomes

$$\text{Find } \boldsymbol{U} \in \mathbb{R}^{N(h)} \text{ such that: } \boldsymbol{AU} = \boldsymbol{b}$$

Where

$$\boldsymbol{A} = \begin{bmatrix} a(\phi_1, \phi_1) & ... & a(\phi_n, \phi_1) \\ ... & ... & ... \\ a(\phi_1, \phi_n) & ... & a(\phi_n, \phi_n) \end{bmatrix} = \begin{bmatrix} a(\phi_1, \phi_1) & ... & a(\phi_1, \phi_n) \\ ... & ... & ... \\ a(\phi_n, \phi_1) & ... & a(\phi_n, \phi_n) \end{bmatrix}$$

$$\boldsymbol{U} = \begin{bmatrix} U_1 & ... & U_{N(h)} \end{bmatrix}^T \qquad \boldsymbol{b} = \begin{bmatrix} l(\phi_1) & ... & l(\phi_{N(h)}) \end{bmatrix}^T$$

Such linear system can be solved quite easily in the case of the Poisson equation because $\boldsymbol{A}$ is non singular and

$$a(\phi_i, \phi_j) = \begin{cases} 4 & \text{if} \quad i = j \\ -1 & \text{if} \quad d(i,j) = h \\ 0 & \text{else} \end{cases}$$

where $d(i,j)$ is the Euclidean distance of the Cartesian coordinates of nodes i and node j. In other words $a(\phi_i, \phi_j)$ returns -1 if the nodes i and j are horizontally or vertically adjacent to each others. Technically we are not required to numerically approximate any integral for constructing the non-singular stiffness matrix $\boldsymbol{A}$ and we could just solve the equivalent finite difference scheme. On the other side the code implemented constructs such matrix using quadrature rules and then such "analytical' result is used to test that the matrix we construct is correct. In order to evaluate $\boldsymbol{b}$ we are required to pick a quadrature rule for each entry $l(\phi_j)$ due to the $f(x,y)$ term. For both the construction of $\boldsymbol{A}$ and $\boldsymbol{b}$ different alternatives quadrature rules have been attempted, the most efficient (in terms of running time and precision) is used in the package which is a triangular quadrature rule described in [3].

2.2 Method of manufactured solutions

In order to test the approximated solution we measure errors by the method of manufactured solutions: we pick a function $u(x,y)$ such that $u=0$ on $\partial\Omega$, solve the Poisson equation 1 this time for $f(x,y)$ and then with such function we apply the finite element method to approximate a solution $u_h(x,y)$. Finally we measure errors with the $L2$-norm

$$\|u(x,y)-u_h(x,y)\|_{L^2(\Omega)} = \sqrt{\int_\Omega |u-u_h|^2} \tag{7}$$

Notice that for the latter we need to integrate over the whole square-domain where a tensor product version the 1D Gauss-Legendre quadrature is used. In order to get correct convergence to the analytical solution we check that by halving the diameter h we decrease such error to approximately a quarter (or less) of the original error, namely that

$$\left\|u-u_{\frac{h}{2}}\right\|_{L^2(\Omega)} \lessapprox \frac{1}{4}\|u-u_h\|_{L^2(\Omega)} \tag{8}$$

.

2.3 Mathematical formulation for Helmholtz

Before describing the code to solve (6) we introduce a variant of the Poisson equation namely the Helmholtz equation with homogeneous Dirichlet boundaries conditions

$$\begin{cases} -\Delta u + cu = f & \text{in} \quad \Omega \\ u = 0 & \text{on} \quad \partial\Omega \end{cases} \tag{9}$$

where $c(x,y) \in L^\infty(\Omega)$. Again we do not go into the theory but we mention class notes by Herve Le Dret [4] where a proof on the variational formulation of problem (9) is given. Using similar arguments as before the system $\boldsymbol{AU}=\boldsymbol{b}$ is such that the i,j-th entry of $\boldsymbol{A}$ is:

$$A_{i,j} = \int_\Omega (\nabla\phi_i \cdot \nabla\phi_j + \phi_i\phi_j c) := a(\phi_i,\phi_j)$$

Differently from before we can't really generalize a result due to the presence of the $c(x,y)$ term, in addition we need to consider that such term does not necessarily vanish for interior nodes "negative diagonally" adjacent; which is when $d(i,j)=\frac{\sqrt{2}}{2}h$ and $x_i<x_j \wedge y_i>y_j$ or $x_i>x_j \wedge y_i<y_j$. The $\boldsymbol{b}$ vector would trivially be equivalent as the Poisson one.
We now describe the code implemented.

3 Elliptic

In this section we describe the code used to generate the triangulation. We try to be as descriptive as possible by writing name of scripts, modules, classes, packages, methods and similar with **bold** font meanwhile we mention inputs, attributes, outputs and similar with *italic* font. Notice that each input of the whole Python package is supported with an exemplified input; if one wants to change inputs it has to assured that the type of input is the same.

3.1 Triangulation

The first step is to generate the triangulation over an arbitrary square-domain. For this purpose we create the class **Elliptic** that, as the name suggests, is helpful for solving, in the domain Ω previously described, Poisson, Helmholtz and possibly more elliptic PDEs. In the package **basis**, in the module **elliptic**, the class **Elliptic** takes three inputs:

1. *nodal_value*: An integer that initialises the number of interior nodes of the triangulation. More specifically the total number of interior nodes of the triangulation is $(nodal\_value - 2)^2$.

2. *length*: A float that would set the length of the square-domain of the problem.

3. *origin*: A list of two floats denoting the Cartesian coordinate of the lower-left vertex of the square-domain where our problem is restricted.

The eventual object of type **Elliptic** will then have the following attributes:

- *nodes*: A np.narray representing an ordered list of the Cartesian coordinates of the interior nodes of the triangulation

- *h*: A float denoting the diameter of the triangles. Such float decreases as we increase the number of interior nodes, more particularly if we want to halve it we increase *nodal_value* to $(2nodal\_value - 1)$.

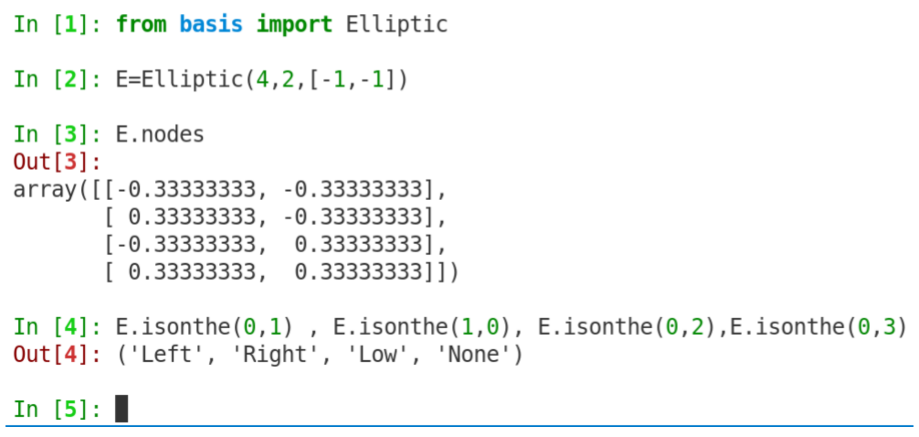
- *length*: The length of the square-domain of the problem.

- *phi*, *phi_x* and *phi_y* : Piecewise linear functions of three variables representing respectively the piecewise linear functions ϕ_i , $\partial\phi_i/\partial x$ and $\partial\phi_i/\partial y$. These functions are imported from the module **Dirichlet** and redefined so that they take as input iterates of *nodes*.

- *data16* and *data46*: Objects of type pandas.core.frame.DataFrame coming from reading respectively **data16.csv** and **data46.csv** from the **data** folder. These two files have information of barycentric coordinates and weights for the quadrature rule over a triangular domain. Such datasets emulate the results of [3].

- *domain*: List of four floats representing the x and y bounds of the square-domain (a, b, c and d if we consider the definition of Ω of the previous chapter).

In addition to the **repr** magic method, the class **Elliptic** has the **isonthe** method that takes as input two integers. Call the integers i and j then the method will give position information of the i-th iterate of *nodes* with respect to the j-th one. A small example is proposed in Figure 2. Notice that for the **isonthe** method we do have possible outputs: {'None' 'Same', 'Right', 'Top', 'TopLeft', 'Left', 'Low', 'LowRight'} but we do NOT have possible outputs for 'TopRight' or 'LowLeft'; this method is implemented in order to help determining the support of the bi-linear form $a(\phi_i, \phi_j)$ for both Poisson and Helmholtz equations.
By looking attentively at Figure 1 we can convince ourselves that if the i-th node is 'TopRight' or 'LowLeft' adjacent to j-th node then $\operatorname{supp}\phi_i \cap \operatorname{supp}\phi_j = \emptyset$ which indeed means that the bilinear form vanishes ie. $a(\phi_i, \phi_j) = 0$. Due to this last statement it would be a "waste" of computational power to approximate integrals involving piecewise linear functions associated to interior nodes "positive diagonally" adjacent. That's why we do not consider the outputs 'TopRight' and 'LowLeft'. We now move to the next object namely the **Dirichlet** module.



```
In [1]: from basis import Elliptic

In [2]: E=Elliptic(4,2,[-1,-1])

In [3]: E.nodes
Out[3]:
array([[-0.33333333, -0.33333333],
       [ 0.33333333, -0.33333333],
       [-0.33333333,  0.33333333],
       [ 0.33333333,  0.33333333]])

In [4]: E.isonthe(0,1) , E.isonthe(1,0), E.isonthe(0,2),E.isonthe(0,3)
Out[4]: ('Left', 'Right', 'Low', 'None')

In [5]: 
```

Figure 2: Screenshot of Python interpreter showing implementation of Elliptic class with $nodal\_value = 4$, $length = 2$, $origin = [-1, -1]$

3.2 Dirichlet

As we mentioned earlier the attributes *phi*, *phi_x* and *phi_y* of the class **Elliptic** are imported from the **Dirichlet** module. In this module there are three functions namely **nodal_basis**, **nodal_basis_x** and **nodal_basis_y**, they take as input:

1. *x, y*: The x and y floats Cartesian coordinate where we want to evaluate our corresponding piecewise linear function ϕ_i, $\partial\phi_i/\partial x$ or $\partial\phi_i/\partial y$.

2. *nodal_point*: A list whose two float entries are Cartesian coordinate of the i-th node whom we associate such piecewise linear function.

3. h: Float measuring the diameter of the triangles of the support of the linear piecewise function.

The output is a float representing the map of the corresponding linear piece-wise function at the point $(x, y) \in \mathbb{R}^2$. The module is called **Dirichlet** because it reflects the fact that these functions satisfy Dirichlet boundaries conditions on the domain Ω presented in the previous chapter. We now describe how the data for the triangular quadrature rule is imported.

4 Data frame

In order to get data we use the **pandas** package together with the **read_csv** function from such package. In order to make sense of it we type ourselves the data set in a **.csv** file. The relevant paper [3] where the data come from offers also good reasoning on how the orbits maps S_{21} and S_{111} are defined which indeed is the logic behind how our data-set is created.

4.1 Triangular quadrature rule

The attributes *data16* and *data46* of **Elliptic** are two **pandas** sets of data. These two data-sets contain triangular barycentric coordinates and weights that are deployed for a triangular quadrature rule. By only knowing the Cartesian coordinates of the vertices of an arbitrary triangle we can convert each barycentric coordinate to its corresponding Cartesian coordinate inside the arbitrary triangle. In the **quadrature** module the **triangle_quadrature_rule** function is primarily responsible for such conversion task.
The **triangle_quadrature_rule** function takes as inputs

- *dataframe*: A pandas DataFrame object, in our case we use the attributes *data16* or *data46* of the **Elliptic** class.
- *function*: The function of two variables we want to integrate. Such function need to be redefined so that it takes as input a two dimensional list instead of the ordinary two variables. This is due to the fact that the conversion from barycentric coordinate gives as output a two dimensional list of floats representing the Cartesian coordinate of the corresponding node of integration.
- *vertex_1*, *vertez_2*, *vertex_3*: Two dimensional lists of floats with Cartesian coordinates of the triangle where we want to integrate our function, in our case we use entries of *nodes*. Vertices coordinates will determine the conversion from barycentric coordinates.
- *base_length*: Float representing the length of the base of the triangle where we want to integrate. Technically we would not require such input. The main idea is that we also need to compute the area of the polygon because it is required in any quadrature rule. For a triangle we could compute the area using the vertices coordinates together with Heron's formula but in our case is slightly more efficient having the *base_length* input so that we can use the h attribute and the fact that the triangles are rectangular/equilateral.

In order to test such quadrature rule there are many examples in **test_2** script. We now move to the **poisson** module, where the **Poisson** class is stored.

5 Poisson

As the name suggests the **Poisson** class solves (up to its abilities) the Poisson approximated problem (6). We want the new class **Poisson** to inherit all the attributes that we previously gave to the **Elliptic** class. For this purpose we make **Poisson** a subclass of **Elliptic**. In addition to the previous inputs of **Elliptic** the inputs of **Poisson** include:

1. f : The $f(x, y)$ function of the Poisson problem written as a function of two variables just like "instinct" may suggest.
2. u : The analytical expected solution $u(x, y)$ written in analogous form as f . Notice we do not need this input to get an approximate solution $u_h(x, y)$, this input is intended only if we wanted to measure errors by the method of manufactured solutions.

The Poisson class has many methods, each of them is preceded by an underscore because they all contribute to a final method which we spoil being

- **uh**: A method of the **Poisson** class that takes as inputs two floats x and y . The method returns the value of the approximation $u_h(x, y)$ of $u(x, y)$ at the point $(x, y) \in \mathbb{R}^2$. If it is the first time we run the method then a single loading bar, using the **tqdm** package, pops in. The loading bar monitors the process of computing $u_h(x, y)$. After completion the resulting float shows. If it is not the first time we run the method then no actual computations is required; no loading bar pops in and the approximated floats returns almost immediately.

Before enlarging the description of the **Poisson** class, we present an implementation the Python interpreter **ipython**.
Suppose we wanted to solve the Poisson equation (1) with

$$\Omega = \{(x, y) \in \mathbb{R}^2 \quad | \quad 0 \leq x, y \leq 100\}$$

Pick for example

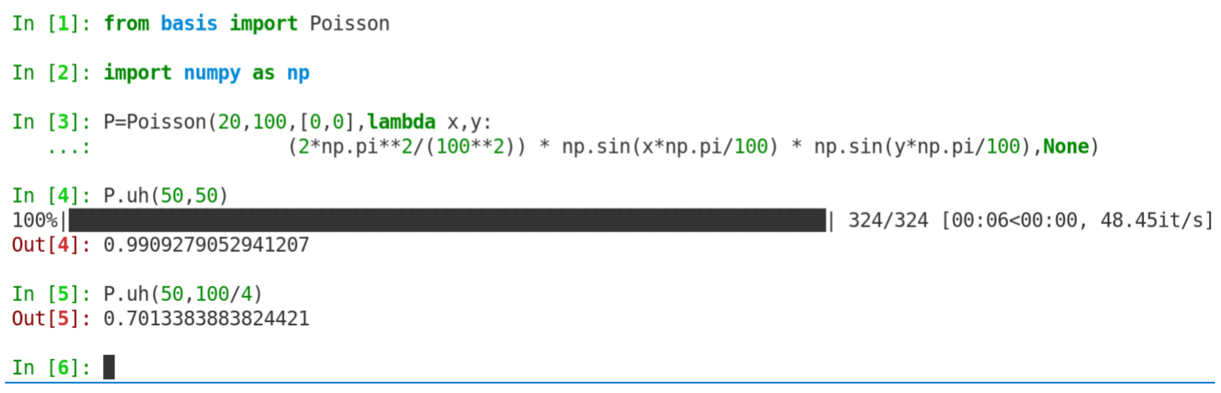
$$u(x, y) = \sin\left(x\frac{\pi}{100}\right)\sin\left(y\frac{\pi}{100}\right)$$

Notice also $u = 0$ on $\partial\Omega$ and if we solve for $f(x, y)$ we get:

$$-\Delta u = -\partial_{xx}u - \partial_{yy}u = \frac{2\pi^2}{100^2}u \implies$$

$$\implies f(x, y) = \frac{2\pi^2}{100^2}\sin\left(x\frac{\pi}{100}\right)\sin\left(y\frac{\pi}{100}\right)$$

Notice $u(50, 50) = 1$ and $u(50, 25) = \sin\left(\frac{\pi}{4}\right) \approx 0.707$.
Conversely in the Python interpreter we give the corresponding inputs for the domain and for $f(x, y)$ (see Figure 3). Finally we approximate a solution $u_h(x, y)$ and notice that $u_h(50, 50) \approx u(50, 50)$ and $u_h(50, 25) \approx u(50, 25)$.



```
In [1]: from basis import Poisson

In [2]: import numpy as np

In [3]: P=Poisson(20,100,[0,0],lambda x,y:
   ...:                 (2*np.pi**2/(100**2)) * np.sin(x*np.pi/100) * np.sin(y*np.pi/100),None)

In [4]: P.uh(50,50)
100%|████████████████████████████████████████████████████████████| 324/324 [00:06<00:00, 48.45it/s]
Out[4]: 0.9909279052941207

In [5]: P.uh(50,100/4)
Out[5]: 0.7013383883824421

In [6]: █
```

Figure 3: Screenshot of Python interpreter showing implementation of Poisson class with *nodal_value* = 20, *length* = 100, *origin* = [0, 0], $f = f(x, y)$, u = None

5.1 Attributes of Poisson

We introduce the attributes of the **Poisson** class.
In addition to the ones already given by **Elliptic**, an eventual **Poisson** object, will have attributes:

- *A*, *b*, *U*, *L2error*: Strings that contains a sentence telling us that the corresponding constructor method would need to be run in order to update the attribute with a numerical value (i.e. array or float).

- *integrand_bilinear_form*: As the name suggests it is a function of 4 variables representing the integrand of the bilinear form $a(\phi_i, \phi_j)$ namely:

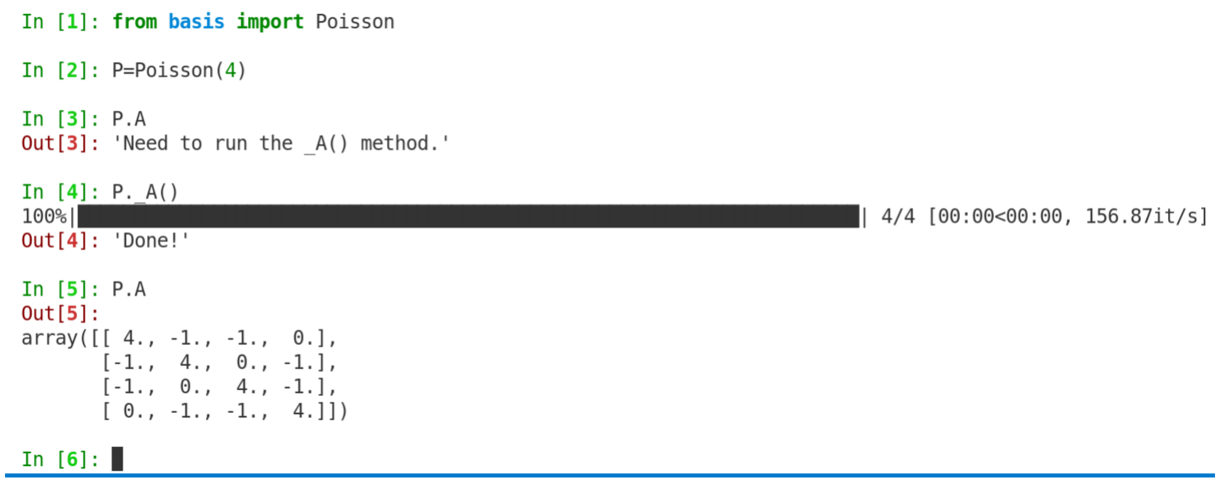
$$\nabla\phi_i(x, y) \cdot \nabla\phi_j(x, y)$$

  .

- *f*, *u*: the $f(x, y)$ and $u(x, y)$ functions.

5.2 Construct the stiffness matrix

Now we are in shape to construct the stiffness matrix $\boldsymbol{A}$ of (6). All we really need is in facts being able to compute the bilinear form $a(\phi_i, \phi_j)$. In the **_a** method we use **triangle_quadrature_rule** and *integrand_bilinear_form* for approximating $a(\phi_i, \phi_j)$ for arbitrary i-th and j-th nodes from *nodes*. Furthermore we use the **isonthe** method for determining where the i-th node is with respect to the j-th one. For all the possible outputs of **isonthe**, the **_a** method integrates on the correct support. If for example the i-th node is on the left of the j-th one then the support of the integrand of $a(\phi_i, \phi_j)$ is the union of the triangles **1** and **6** of the i-th node (recall Figure 1), or equivalently the triangles **3** and **4** of the j-th node and so on for different examples. The **_a** method breaks this problem in a quite hard-coding way; we admit that a different way could probably be implemented.
The **_A** method initialises an empty numpy array and updates each entry by using the **_a** method. The **tqdm** package is used in order to get the loading bar. Once the computational process is complete the attribute A of the **Poisson** object is updated with the corresponding completed numpy array. An example in **ipython** is given in Figure 4 . In **test_1** script we



```
In [1]: from basis import Poisson

In [2]: P=Poisson(4)

In [3]: P.A
Out[3]: 'Need to run the _A() method.'

In [4]: P._A()
100%|██████████████████████████████████████████████████| 4/4 [00:00<00:00, 156.87it/s]
Out[4]: 'Done!'

In [5]: P.A
Out[5]:
array([[ 4., -1., -1.,  0.],
       [-1.,  4.,  0., -1.],
       [-1.,  0.,  4., -1.],
       [ 0., -1., -1.,  4.]])

In [6]: 
```

Figure 4: Screenshot of Python interpreter showing implementation of Python class and _A() method with $nodal\_value = 4$

build the stiffness matrix of the Poisson equation for different examples and we compare it with the expected result. We see that the precision is up to the 15-th decimal when using the data coming from *data16*. This means we are not required to use higher precision data such as *data46*. This reflects the fact that the integrand is piecewise-constant.

5.3 Construct b

Now we construct the vector $\boldsymbol{b}$. We only need to be able to approximate $l(\phi_j)$ of (6). The **_l** method of the **Poisson** class takes care of this task. Differently from before we do not use **isonthe** method because the support of the integrand of the linear form is the whole support of ϕ_j for all j. We do prefer more precision data such as *data46* because we do not know what kind of function $f(x, y)$ is. The **_b** method shows a loading bar while using the **_l** method for each element in *nodes*. After completion it updates the b attribute with the corresponding numpy array and shows a single loading bar for the computational process.

5.4 Assembly

We can now solve the linear system (6) using the two updated attributes A and b. If one of the two attributes A or b is still a string then the method **_U** shows a loading bar and couples the tasks previously described for **_A** and **_b** while showing a single loading bar. Reason why the two tasks are not simply recalled and are somehow coupled is because we want to show only one loading bar and not two. If both the attributes A and b are not string then the method **_U** solves the linear system using the scipy's package linalg. The command to solve the linear system is simple but efficient. It will be more problematic if by any chance the resulting stiffness matrix is singular but it can never happen in the case of the Poisson equation. Once the linear system is solved we update the attribute U of the Poisson class with the corresponding solution of the system. Finally in the **uh** method we introduce the inputs x and y to make it look more like a function instead as a method.
If the U attribute is not a string then **uh** method returns the sum of the product between each entry of U with the attribute function *phi* evaluate at each node of *nodes* and at the input Cartesian coordinates x and y. Notice this is equivalent as returning

$$u_h(x,y) = \sum_{i=1}^{N(h)} U_i \phi_i(x,y)$$

If the attribute is still a string the **_U** method is recalled in order to update with the correct array.
This completes the implementation of the finite element method with Python of piecewise linear basis functions to solve the Poisson equation on a square domain. The example of Figure 3 can again serve as an example of how to implement the class

5.5 Method of manufactured solutions

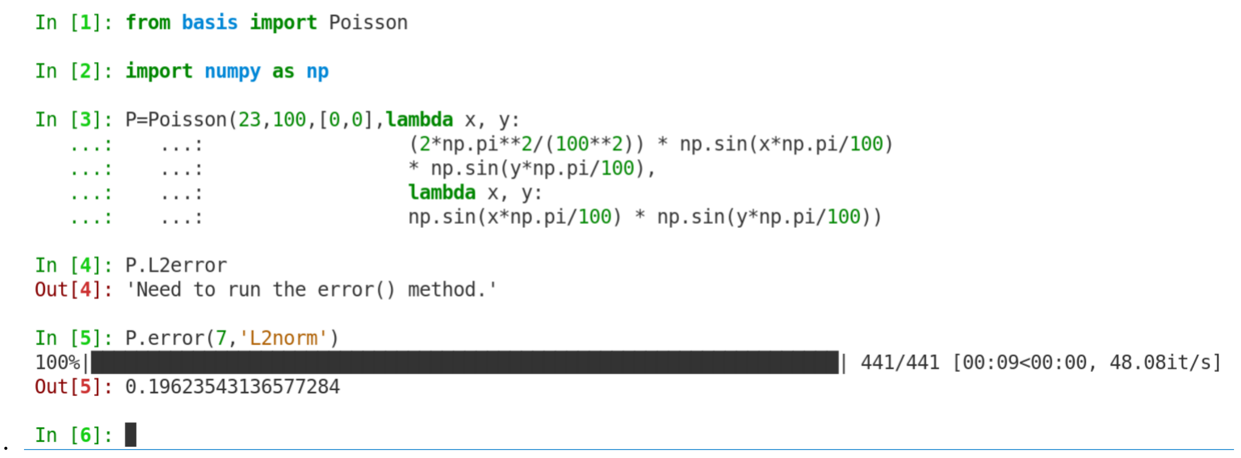
In order to test whether the results are correct or not we do not offer only the example in Figure 3. For such task the script **test_3** takes as input three examples and makes sure that the $L2$ norm of the difference function between a known solution $u(x,y)$ of equation (1) and its approximation $u_h(x,y)$ satisfies equation (8). In addition we make sure that by halving again the "halved diameter" the equation would still be satisfied. Notice that in order to half the diameter of triangulation whose initial input is *nodal_value* then our desired halved diameter triangulation would need to have as input 2*nodal_value - 1*. In order to evaluate the error, which we recall being

$$\|u - u_h\|_{L^2(\Omega)}$$

we need to approximate an integral and, as mentioned early in this report, we use a tensor product variant of the 1D Gauss-Legendre quadrature rule.
In the **quadrature** module the function **GaussLegendre** takes as input

1. f: The two dimensional function whose integral we want to approximate
2. a, b, c, d: The x and y bounds of the square regions where we want to approximate the integral.
3. n: The desired degree of the Legendre polynomial of the Gauss Legendre quadrature rule. The higher the value, the slower but more precise the approximation.

The output is the approximation of such integral. Going further in the **quadrature** script we find the short **L2error** function that simply computes the **GaussLegendre** approximation of the $L2$ norm of two input functions namely in the code called u and uh. Such function is used in the final method of the **Poisson** class namely **error** that returns the error we seek. Notice that such method will use the eventual attribute u of the **Poisson** class so we need to make sure we know the analytical solution $u(x, y)$ and we properly define it when initialising the class. As usual we propose a **ipython** example in Figure 5; we use the same $f(x, y)$ function as in Figure 3 but now we also give as input the corresponding analytical solution $u(x, y)$. Finally we recall that **test_3** script contains relevant tests following the idea of the



```
In [1]: from basis import Poisson

In [2]: import numpy as np

In [3]: P=Poisson(23,100,[0,0],lambda x, y:
   ...:    ...:                  (2*np.pi**2/(100**2)) * np.sin(x*np.pi/100)
   ...:    ...:                  * np.sin(y*np.pi/100),
   ...:    ...:                  lambda x, y:
   ...:    ...:                  np.sin(x*np.pi/100) * np.sin(y*np.pi/100))

In [4]: P.L2error
Out[4]: 'Need to run the error() method.'

In [5]: P.error(7,'L2norm')
100%|████████████████████████████████████████████████████████| 441/441 [00:09<00:00, 48.08it/s]
Out[5]: 0.19623543136577284

In [6]: ▌
```

Figure 5: Screenshot of Python interpreter showing implementation of Python class and _error() method

method of manufactured solutions.

6 Helmholtz

In the **helmholtz** module the **Helmholtz** class solves the approximated problem of the Helmholtz equation (9) with homogeneous Dirichlet boundaries conditions described at the end of Chapter 2.

6.1 Class for solving Helmholtz

We want to start from something similar as the **Poisson** class meaning that we want to define **Helmholtz** class as a subclass of **Poisson**.
An eventual object of class **Helmholtz** has, in addition to the **Poisson** and **Elliptic** inputs, the following input:

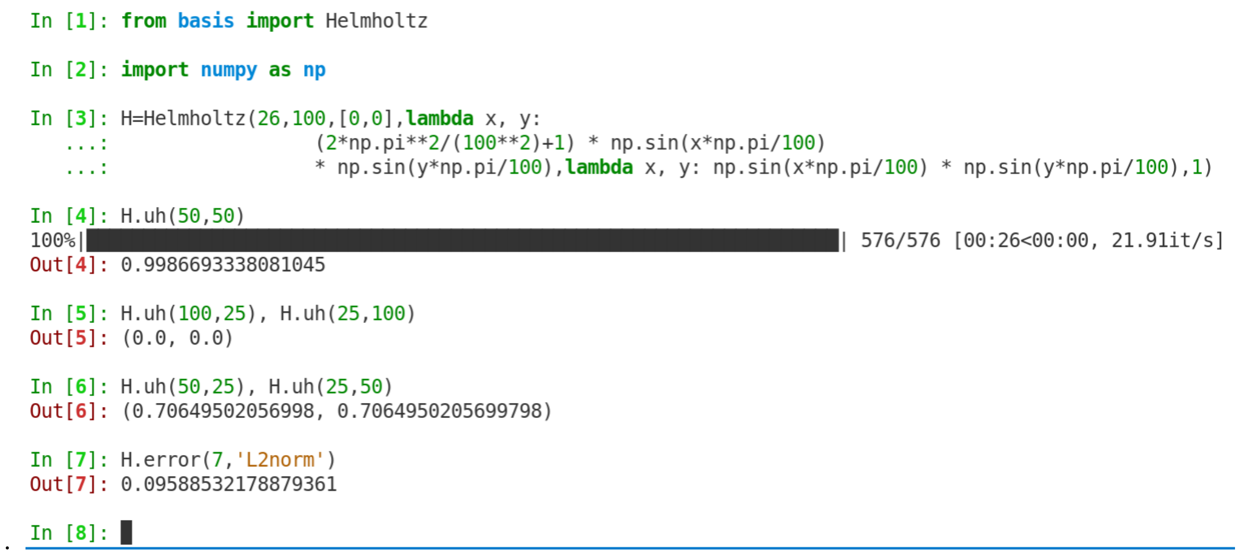
1. c: The function $c(x, y)$ of equation (9)

Considering **Helmholtz** is subclass of **Poisson** most of the wheels we need are already there. As we mentioned early in the paper the bilinear form is not the same as before. For this reason, beside adding the attribute c we also redefine the attribute *integrand_bilinear_form* so that it emulates the integrand of the new bilinear form namely

$$\nabla \phi_i \cdot \nabla \phi_j + \phi_i \phi_j c$$

We are nearly done, indeed what could be tricky to forget (because we did notice it only after a week) is that we now also have to redefine the **_a** method because when **isonthe** returns

'TopLeft' or 'LowRight' the bilinear form won't vanish. Moreover it is more convenient (at the cost of higher running time) to use data coming from *data46* because the $c(x, y)$ function is not necessarily a constant function. We present an **ipython** example in Figure 6 where $c(x, y) = 1$ and the $f(x, y)$ function is chosen so that the analytical solution $u(x, y)$ is the same as in Figure 3. In a similar manner as we done in **test_3** we test the **Helmholtz** class



```
In [1]: from basis import Helmholtz

In [2]: import numpy as np

In [3]: H=Helmholtz(26,100,[0,0],lambda x, y:
   ...:                  (2*np.pi**2/(100**2)+1) * np.sin(x*np.pi/100)
   ...:                  * np.sin(y*np.pi/100),lambda x, y: np.sin(x*np.pi/100) * np.sin(y*np.pi/100),1)

In [4]: H.uh(50,50)
100%|████████████████████████████████████████████████████████████| 576/576 [00:26<00:00, 21.91it/s]
Out[4]: 0.9986693338081045

In [5]: H.uh(100,25), H.uh(25,100)
Out[5]: (0.0, 0.0)

In [6]: H.uh(50,25), H.uh(25,50)
Out[6]: (0.70649502056998, 0.7064950205699798)

In [7]: H.error(7,'L2norm')
Out[7]: 0.09588532178879361

In [8]:
```

Figure 6: Screenshot of Python interpreter showing implementation of Helmholtz class and _error() method

in **test_4**.

7 Conclusion

We hope this report will facilitate understanding the code of the Python package. We mention that this last, apparently small, fact of the different supports between the two **_a** methods of the two classes had been realized after many days and slowed the assessment but luckily we managed to fix it. We would like to thanks the supervisor of the candidate but due to anonymity requirement of this report we can't.

References

[1] Suli E., 2020, Lecture Notes on Finite Element Methods for Partial Differential Equations, University of Oxford, Mathematical Institute.

[2] Farrell P., 2018 Finite Element Methods for PDEs, University of Oxford, Mathematical Institute

[3] Zhang L., Cui T. Liu H. ,2009, A Set Of Symmetric Quadrature Rules On Triangles And Tetrahedra , LSEC, ICMSEC, Academy of Mathematics and Systems Science, Chinese Academy of Sciences, Beijing 100190, China

[4] Herve Le Dret, 2011-2012, Class Notes M1 Mathematics, MM26E Numerical Approximation of PDEs,